AFRL-IF-RS-TR-2001-18
Final Technical Report
February 2001

COMPILING SCIENTIFIC PROGRAMS FOR SCALABLE PARALLEL SYSTEMS

Rice Universtiy

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D515

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2001-18 has been reviewed and is approved for publication.

APPROVED: [illegible]

JOSEPH A. CAROZZONI
Project Engineer

FOR THE DIRECTOR: [illegible]

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

COMPILING SCIENTIFIC PROGRAMS FOR SCALABLE PARALLEL SYSTEMS

Ken Kennedy, John Mellor-Crummey,
Vikram Adve, Robert J. Fowler,
And Guohua Jin

Contractor:	Rice University
Contract Number:	F30602-96-1-0159
Effective Date of Contract:	1 June 1996
Contract Expiration Date:	31 May 1999
Program Code Number:	C276
Short Title of Work:	Compiling Scientific Programs for Scalable Parallel Systems
Period of Work Covered:	Jun 96 - May 99

Principal Investigator:	John Mellor-Crummey
Phone:	(713) 527-4820
AFRL Project Engineer:	Joseph A. Carozzoni
Phone:	(315) 330-7796

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Joseph A. Carozzoni, AFRL/IFTB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** FEBRUARY 2001	**3. REPORT TYPE AND DATES COVERED** Final Jun 96 - May 99

4. TITLE AND SUBTITLE COMPILING SCIENTIFIC PROGRAMS FOR SCALABLE PARALLEL SYSTEMS	**5. FUNDING NUMBERS** C - F30602-96-1-0159 PE - 62301E PR - C322 TA - 00 WU - 02
6. AUTHOR(S) Ken Kennedy, John Mellor-Crummey, Vikram Adve, Robert J. Fowler, and Guohua Jin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rice University P.O. Box 1892 - MS 16 Houston TX 77251-1892	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 N. Fairfax Drive Arlington VA 22203; Air Force Research Laboratory/IFTB 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2001-18

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Joseph A. Carozzoni/IFTB/(315) 330-7796

12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This report details research into compiler technology to support machine-independent data parallel programming for scientific application. The investigation focused on design and development of dHPF, an advanced prototype compiler for High Performance FORTRAN (HPF). The research performed in this project included new techniques for recognizing implicit parallelism in sequential programs, a powerful and precise set-based framework for analysis and transformation of data-parallel programs, support for sophisticated data and computation partitioning, effective support for generation and optimization of parallel code for message-passing and distribution shared-memory systems; techniques for supporting sophisticated data-parallel applications with irregular structure, and an integrated compiled effort that included a multi-level memory hierarchy simulator and techniques for improving locality in irregular computations.

14. SUBJECT TERMS Data Parallel Programming, High-Performance FORTRAN, Memory Hierarchy Simulator	**15. NUMBER OF PAGES** 88
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Contents

1 Introduction **1**

2 Approach **2**

3 Accomplishments **3**

3.1 Detection of Implicit Parallelism . . . 3
3.2 Computation Partitioning Model . . . 6
3.3 An Integer Set Framework for Program Analysis and Optimization . . . 7
3.3.1 Description of the Set Framework for Data-Parallel Compilation . . . 9
3.3.2 Optimizations using the Integer Set Framework . . . 11
3.3.3 Extensions for Symbolic Distribution Parameters . . . 13
3.3.4 An Optimized Virtual Processor Model . . . 15
3.3.5 Code Generation using Virtual Processors . . . 17
3.4 Computation Partitioning Optimizations . . . 18
3.4.1 Computation Partitioning for Reductions . . . 18
3.4.2 Parallelizing Computations that use Privatizable Arrays . . . 19
3.4.3 Partial Replication of Computation . . . 21
3.5 Code Generation for General Computation Partitions . . . 22
3.5.1 Realizing Computation Partitions . . . 23
3.5.2 Constraint Propagation and Control Flow Simplification . . . 27
3.6 Experimental Evaluation of dHPF Compilation Techniques . . . 31
3.6.1 Compiler Performance . . . 32
3.6.2 Performance of dHPF-generated code . . . 33
3.6.3 SP . . . 35
3.6.4 BT . . . 38
3.7 Compiler-Support for Advanced Partitionings . . . 40
3.7.1 Multipartitioning in dHPF . . . 41
3.7.2 Evaluation of Compiler-Support for Multipartitioning . . . 43
3.8 Using Data-Parallel Languages for Irregular Applications . . . 46
3.8.1 Hierarchical Methods . . . 46
3.9 Implementation Comparison . . . 47
3.9.1 The Hand-coded Implementation . . . 48
3.9.2 The HPF Implementation . . . 49
3.9.3 Initial HPF vs. MPI Performance . . . 49
3.9.4 Rationale for Performance Differences . . . 50
3.9.5 Improving HPF Performance . . . 51
3.9.6 Discussion . . . 52
3.10 Compiler and Run-time Support for Software Distributed Shared-Memory . . . 54
3.10.1 Effectiveness of Integrated Compiler and Run-time Optimizations . . . 56
3.11 Integrated Compiler Support for Tools . . . 57
3.12 Other Accomplishments . . . 59
3.13 Evaluation of Accomplishments . . . 60

4 Project Publications **61**

5 Personnel **63**
5.1 Faculty 63
5.2 Research Staff 63
5.3 Graduate Students 64
5.4 Undergraduate Students 64

6 Interactions **65**
6.1 Conference Presentations 65

List of Figures

1 Primitive sets and mappings for compiling data-parallel programs. 9
2 Construction of primitive sets and mappings for an example program. (Align$_A$, Align$_B$, and Dist$_T$ also include constraints for the array and template ranges, but these have been omitted here for brevity.) 9
3 Equations for computing communication sets 10
4 Loop splitting to overlap communication and computation. 14
5 Active virtual processors for computing, sending and receiving 15
6 Code generation for SEND with optimized virtual processor model 16
7 Computation partitioning for a loop nest from subroutine `lhsy` of the NAS SP computational fluid dynamics benchmark. 19
8 Using LOCALIZE to partially replicate computation in subroutine `compute_rhs` from NAS SP. 22
9 Constructing a code template from iteration spaces. 24
10 Example showing iteration sets constructed for code generation. 24
11 HPF source fragment abstracted from the Erlebacher benchmark. 25
12 Skeletal SPMD code for Fig. 11 with partitioned computation. 25
13 Skeletal SPMD code for Fig. 12 after simplification. 30
14 Space-time diagram of hand-coded MPI for SP (16 processors). 37
15 Space-time diagram of dHPF-generated MPI for NAS SP application benchmark (16 processors). 37
16 Space-time diagram of hand-coded MPI for NAS BT application benchmark (16 processors). 39
17 Space-time diagram of dHPF-generated MPI for NAS BT application benchmark (16 processors). 40
18 3D Multipartitioning on 16 processors. 41
19 dHPF-generated NAS SP using 3D multipartitioning. 44
20 Speedups for MPI hand-coded multipartitioning and dHPF-generated multipartitioning versions of NAS SP benchmark (class A). 44

List of Tables

Table 1:	Breakdown of dHPF compilation time	33
Table 2:	Comparison between hand-written MPI, dHPF and pgppg for SP	36
Table 3:	Comparison between hand-written MPI vs. dHPF and PGI	39
Table 4:	Ratio of performance metrics for single-processor executions	45
Table 5:	Key for row heading abbreviations in performance tables	56
Table 6:	NAS-BT class A 16 columnwise computation partitioning	56
Table 7:	NAS-BT class A 4x4 computation partitioning.	57

1 Introduction

Since the inception of scalable parallel computation, the critical impediment to widespread acceptance has been the absence of a high-level programming model for development of applications that can run with high performance across the entire spectrum of parallel architectures. Fully automatic parallelization has failed to materialize for configurations beyond small-scale shared-memory processors. Distributed shared memory, while promising, has not yet been able to deliver scalable performance on the full spectrum of machine configurations and applications. Currently, the most common approach to portable parallelization is the use of a standard message-passing library, such as MPI. This is common even for programs developed on hardware distributed shared memory systems. Unfortunately, the message-passing programming model puts an enormous burden on the application developer, who must not only design the parallelization strategy but must handle the details of communication and memory management. If no higher-level programming interface is successful, the limitations of message-passing are likely to inhibit the widespread use of scalable parallelism, a technology that is critical to the defense enterprise as well as to the solution of major national problems in science, engineering, and commerce.

High Performance Fortran (HPF) [Hig93, KLS$^+$94], developed in the first half of this decade represents an attempt to harness the power of automatic parallelization technology to provide a commercially-viable high level programming model for developing fully portable programs. HPF provides an attractive model for parallel programming because of its relative simplicity. Principally, programmers write a single-threaded Fortran program and use data layout directives to map data elements onto an array of processors. HPF compilers use these directives to partition a program's computation among processors, and to synthesize necessary data movement and synchronization. As of 1997, fifteen companies were offering HPF compilers, including all major parallel computer vendors, and nearly forty major applications had been developed in the language, some over 100,000 lines. In addition, several of the DOD Modernization CHESSI efforts planned to use HPF in their applications.

Although there was initally considerable enthusiasm for HPF, it failed to achieve widespread acceptance by scientists as the model of choice for developing parallel applications. The success of HPF has been principally limited by the shortcomings of its compilers which were adapted from technology for automatic parallelizers of the late 80s and early 90s. This technology has just not been sophisticated enough to deliver scalable performance across a variety of architectures and applications. Compilation techniques in use by commercial HPF compilers fail to generate code that achieves performance competitive with that of hand-coded programs for typical applications. As a result, many applications developers have been reluctant to use HPF, choosing to suffer the burdens of message-passing instead. Other developers have simply delayed the transition to scalable parallelism.

Second, it is difficult for developers to use HPF compilers to parallelize existing codes. Commercial HPF compilers lack analysis and code generation capabilities to effectively parallelize a wide range of loops written in a Fortran 77 style. Therefore, for codes to achieve reasonable parallelism when compiled with commercial HPF compilers, they need to be rewritten substantially.

Third, the acceptance of HPF has been hampered by its lack of support for irregular, adaptive codes that dominate many fields of scientific computation, such as engineering design. Although the HPF Forum added support for irregular mappings to the language, no commercial compiler has supported this critical feature. Furthermore, even earlier research compilers have been unable to integrate support for irregular and regular meshes in a manner that was robust enough for real applications.

Finally, developers have been slow to adopt HPF because of the lack of programming tools that enable programmers to develop, debug and tune their programs entirely at the source level. Since HPF compilers perform radical program transformations, effective tools for working with HPF programs require extensive compiler support.

The goal of this project was to overcome those difficulties by developing a whole new generation of technologies that are powerful enough to eliminate the previous shortcomings of HPF compilers and yet practical enough to be used in commercial products.

2 Approach

The project's research focused on developing compiler technology for High Performance Fortran (HPF). The project's investigation of compiler technology to support data-parallel programming centered around the design and development of dHPF, an advanced prototype compiler for High Performance Fortran.

A principal focus of the dHPF compiler research was on compiler support for data and computation partitioning. The strategy used to partition an application's data and computation plays a critical role in determining the application's performance. The partitioning used determines the application's possible parallelizations, which in turn determines the maximum attainable parallel efficiency. To support effective parallelization of a broader class of computations with dHPF, we developed a new computation partitioning framework that enables processors to compute values for any data, independent of whether or not the data is mapped to that processor. This framework supports a much more general class of partitionings than current commercial or other research compilers. Also, the framework provides the capability to experiment with different algorithms for selecting computation partitionings.

Supporting these more general partitionings required new techniques for communication analysis and code generation. To make communication analysis and code generation for our general computation partitioning model tractable, as well as support advanced optimizations, a high-level abstract model is necessary. Accordingly, we designed the dHPF compiler around a powerful integer-set framework that performs communication analysis and code generation by manipulating sets of and mappings between integer tuples that represent data elements, iterations, and processors. This set framework enables advanced optimizations to be expressed simply and concisely, yet generally, in terms of set equations. Used in conjunction with data dependence analysis, this framework provides a basis for determining precisely what values need to be communicated, and reasoning about closed-form representations of these communication sets. Manipulation of these communication sets and iteration spaces enables sophisticated optimizations such as loop splitting to overlap communication and computation, and communication blocking strategies such as coarse-grain pipelining, which sacrifices potential parallelism to reduce communication overhead by increasing communication granularity.

An important goal in the design of the of the dHPF compiler was to develop analysis algorithms that are abstract and machine-independent, so that a single compiler framework can serve all platforms, and any optimization can be applied to any architecture (where profitable).

The project research also aimed to abstract the notion of distribution so that a single concept can handle both regular and irregular mappings, and standard compiler optimizations for static computation partitioning, communication vectorization, and latency hiding can be applied to irregular as well as regular applications.

Finally, a component of the dHPF project was to develop new compiler technology to support source-level programming tools. The project has developed techniques to track compiler transformations, describe them to programmers, and relate measured performance information back to the source program. During compilation, dHPF automatically records program transformations both in terms of detailed low-level changes and in terms of higher-level, semantically meaningful transformations. These are kept in a program database along with detailed compiler analysis results and optimization decisions. This information repository provides a basis for constructing programming tools that provide detailed source-level feedback to programmers.

3 Accomplishments

The principal accomplishment of this project has been the development of compiler analysis and code generation technology to help machine-independent, data-parallel programs to achieve high performance on a range of scalable parallel architectures for a broad spectrum of scientific applications. In the following sections, we provide a brief overview of our key technical accomplishments.

3.1 Detection of Implicit Parallelism

One can parallelize an application written in sequential Fortran simply by adding HPF directives to the program and leaving it to an optimizing compiler to analyze, understand and exploit parallelism inherent in the program. However, for this approach to be broadly effective for scientific programs, HPF compilers must be able to effectively parallelize loops with carried data dependences (simply, this means loops that update values that are later used in subsequent iterations).

A particularly important class of loops that carry data dependences are those that compute reductions. A reduction is a commutative and associative operation that maps an array of n dimensions to an array of m dimensions, where $0 \leq m < n$[1]. Reduction operations appear in many contexts including kernels for matrix multiplication, image processing, computational geometry algorithms, sorting, and are commonly used to test for convergence of iterative algorithms.

When compiling programs in which reductions are coded implicitly, an optimizing compiler without explicit support for reduction recognition will naively classify reductions as sequential operations. This happens because data dependence analysis will identify the reduction's repeated updates of the same location as an obstacle to parallel execution. If implicit reductions are not recognized and parallelized, they can dramatically degrade parallel program performance. However, reduction computations can be parallelized since commutative updates to an accumulator can be reordered safely.

Reduction computations can be optimized effectively for a variety of architectures. On MIMD parallel computers, a reduction can be computed efficiently by having each processor (in parallel) compute a partial reduction result and then combining these partial results into a final result by using one or more collective communication operations. In this manner, a parallel reduction of n data elements into m data elements on p processors can be computed in time $O(n/p + m \log p)$ (assuming that the data is evenly distributed). To generate reduction code that achieves high performance on distributed-memory machines, it is important minimize communication which can be costly on such systems.

Reduction recognition in the dHPF compiler grew out of earlier work in PFC [Dar86], which recognized implicit SUM and PRODUCT reductions to convert them into equivalent explicit FORTRAN 90 intrinsic calls. For the dHPF compiler, we extended this work to handle more general forms of SUM and PRODUCT reductions, developed support for recognition of MIN, MAX, MINLOC, and MAXLOC reductions, and developed code generation support for distributed memory machines.

[1]An array of zero dimensions represents a scalar value.

Our new approach to recognizing reductions recognizes a variety of multi-level reductions intermixed with other computation. We use a combination of control and data dependence analysis along with pattern matching to recognize reductions at multiple levels inside imperfectly nested loop nests that may contain conditional control flow. Our experiments show that our strategy is effective and exceeds the capabilities of other compilers.

Here we briefly describe how dHPF recognizes SUM, PRODUCT, MIN, MAX, MINLOC and MAXLOC reduction patterns. There are two phases to reduction recognition. First, for each assignment statement, we check if it is reducible and if so, classify its reduction type. Second, we gather related reduction statements into groups. For each group, we check if it is a reducible group (for example, all the statements in the group must have the same reduction type), and decide the levels that the reduction can be carried on.

Single Statement Reductions To identify whether an assignment statement is part of a reduction computation, we use four program representations: an abstract syntax tree (AST), a data dependence graph [KA97], static single assignment (SSA) form [CFR+91] and a control dependence graph [CFR+91]. We first identify assignment statements with incident *true*, *anti* and *output* data dependences as reduction candidates. Next, we use SSA to locate definitions of the right hand side variables and use the AST to inspect the operations on right hand side variables in reduction candidates. For extreme value reductions formed by MIN, MAX, MINLOC, or MAXLOC operations, we use control dependence information to find a control dependence predecessor and compare the control statement with the dependent statements to determine if they form an extreme value reduction. The use of control dependence information distinguishes our method from others, and enables us to recognize extreme value reductions using various control flow structures.

Reduction Groups For SUM or PRODUCT reductions, a reduction group is formed by reduction statements with the same left hand side accumulator and same reduction operator. For example, in the following code fragment

```
do i = lb, ub
  T = T + Y(i)
  T = T + Z(i, n)
enddo
```

both of the assignments to T belong to the same SUM reduction group with the accumulator of T. For MIN/MAX reductions, a reduction group is formed by the assignment statement that records the extreme value along with any assignment statements that record the position of the extreme value.

All statements in a reduction group use the same storage for accumulating the local reduction results (a local accumulator for T in the example above), and use a single collective communication operation to compute the global results. Our approach differs from the reduction handling in the IBM HPF compiler [SKN96]. They first generate a separate communication event for each reduction statement, and then apply reduction coalescing and aggregation to merge communication operations where possible. By forming a reduction group and generating one communication event for each group, we avoid the need for communication coalescing in most cases.

For any pair of statements to be members of the same reduction group, their common accumulator must not be modified or referenced by non-reduction statements at their deepest common loop level.

Reduction Levels The dHPF compiler recognizes multi-level reduction groups. Statements in a reduction group need not all be at the same loop nesting level. We use the data dependence graph to detect modifications and references to a reduction accumulator by non-reduction statements to decide how many loop levels can participate in a reduction. For example, the following loop nest

```
     do k = 1, ub1
S1     Q(k) = S+9
S2     S = S+B(k)
       do i = 1, ub2
G1       S = S+C(i)
         do j = 1, upb3
G1         S = S+E(j)
         enddo
G1       S = S+D(i)
       enddo
     enddo
```

contains one reduction group G_1 that contains the statements as labeled. Since S_1, a non-reduction statement, references the value of S at the same loop level as reduction candidate S_2, S_2 cannot be a member of any reduction group. However, since the intermediate values of S are not referenced by a non-reduction candidate inside the i or j loops, all of the statements marked G_1 are collected into the same reduction group. The reduction for these inner two loops can be computed in parallel to provide the value of S used by S_1. We say group G_1 is reducible in the inner two loops.

Building Reduction Groups To build reduction groups, we perform the following steps:

1. Build reduction groups for statements directly inside a loop.

   (a) Identify assignment statements that meet the criteria for being a reduction candidate. Classify the type of the reduction by examining the contributing operators and the control dependences.

   (b) For each set of reduction candidates that have the same accumulator and reduction type, collect them into a reduction group as long as any other uses or modifications of the accumulator occur outside the innermost loop level at which the reduction dependences are carried.

2. Merge reduction groups at different loop levels. If there are two reduction groups, G_1 at level m and G_2 at level n, where $m < n$, merge G_1 and G_2 if they have the same accumulator and reduction type and no non-reduction candidate accesses the accumulator at levels m through n, inclusive. Otherwise, if the reduction accumulator is accessed at level k, where $m \leq k < n$, group G_1 in the outer loop is not reducible and we cannot merge the two groups.

   To compute whether groups can be merged, for each reduction group G_1 at level m:

   (a) Search all of the dependences of the accumulation variable of G_1 to find other statements that reference or modify it. If one such statement is in another reduction group G_2 which has the same accumulator and reduction operator and it is at some level $k \geq m$, we say G_1 and G_2 are compatible and mark G_2 as a candidate group to be merged with G_1 later. If the statement is not in another compatible reduction group, and it is at some level $k \geq m$, mark G_1 as "not reducible".

   (b) If G_1 is not marked as "not reducible", merge G_1 with all the compatible reduction groups at levels $k \geq m$ (such as G_2 in the above example).

(c) Decide the outermost level at which the reduction can be performed based on the dependences checked in step 2a.

Idioms Recognized The algorithms described above can recognize a broad range of reduction operations. Here we provide a brief enumeration of the reduction features that dHPF can recognize.

- Scalar reductions which accumulate the results into a scalar variable.
- Multi-element array reductions which accumulate reduction results into one or more elements in an array.
- Reduction groups that may contain multiple reduction statements (each of which may be at a different loop nesting level) sharing the same accumulator.
- Reduction operations that are control dependent on conditionals such as

  `if` α `then s = s * a(i)`

  can form a reduction group with other statements outside the control statement.
- MIN/MAX and MINLOC/MAXLOC reductions using different forms of `if` structures. Absolute value operations are fully supported for MIN/MAX or MINLOC/MAXLOC operations, such as those that occur in the SPEC92 TOMCATV benchmark program.
- Reductions closely intermixed with other computations. For example, dHPF can recognize reductions that use arrays or privatizable variables which are defined previously in the same loop; other compilers require that reduction operations be isolated from other computations and that their operands be *prefetchable* [SKN96].

3.2 Computation Partitioning Model

A computation partitioning (CP) for a statement specifies which processor(s) must execute each dynamic instance of the statement. To support effective parallelization of a broad class of computations, we developed a new computation partitioning framework for dHPF that enables processors to compute values for any data, independent of whether or not the data is mapped to that processor. Within dHPF, this enables us to choose an appropriate partitioning well-suited to an application's needs with the aim of maximizing parallelizm and minimizing communication. In this section, we briefly describe the computation partitioning model supported by dHPF.

Research and commercial HPF compilers primarily use the *owner-computes* rule [RP89] to assign CPs to statements. This rule specifies that a computation is executed by the owner of the value being computed. This rule, as well as other variants used in some compilers (e.g., decHPF [HBB+95] and SUIF [AL93]) can be expressed in terms of the processor(s) that own a particular set of data elements. In particular, for a statement enclosed in a loop nest with index vector $\underline{i}$, and for some variable A, the CP $\texttt{on\_home}\{A(f(\underline{i}))\}$, specifies that the dynamic instance of the statement in iteration $\underline{i}$ will be executed by the processor(s) that own the array element(s) $A(f(\underline{i}))$. This set of processors is uniquely specified by subscript vector $f(\underline{i})$ and the layout of array A at that point in the execution of the program. The SUIF compiler [AL93] further restricts all statements in a loop to have the same computation partition. The dHPF compiler supports a more general CP model in which a CP for a statement can be specified as the owner of one *or more* arbitrary data references, and each statement in a program may have its own CP. A statement's CP is specified by a union of one or more `on_home` terms: $CP : \cup_{j=1}^{j=n} \texttt{on\_home}\{A_j(f_j(\underline{i}))\}$. This *implicit* representation of a computation partitioning in dHPF supports arbitrary index expressions or any set of values in each index position in $f_j(\underline{i})$.

We convert the implicit CP form into an explicit integer tuple mapping, CPMap. This is possible when each subscript expression in $f_j(\underline{i})$ is an affine expression of the index variables, $\underline{i}$, with known constant coefficients, or is a strided range specifiable by a triplet *lb:ub:step* with known constant *step*. The overall mapping is a union of mappings for the individual **on_home** terms:

$$\mathrm{CPMap} = \bigcup_{j=1}^{j=n} (\mathrm{Layout}_{\mathrm{A_j}} \circ \mathrm{Ref}_{\mathrm{j}}^{-1}) \bigcap\nolimits_{\mathrm{range}} \mathrm{loop}$$

Here, the mapping for a single term **on_home**$\{A_j(f_j(\underline{i}))\}$ is a composition of the layout and reference mappings, restricted in range to the loop index space. CPMap explicitly specifies the processor assignment for the instance of a statement in loop iteration $\underline{i}$.

When using such a general computation partitioning model that enables each statement to specify which processors need to participate in its execution, control flow statements require careful handling to ensure that each statement is reached by all of the processors that need to execute it. A valid computation partitioning is achieved through a sequence of steps. First, dHPF selects a computation partitioning (CP) for each assignment statement, except for assignments to privatizable variables. For each such statement, the CP is generally chosen to be ON_HOME of one of the references in the statement. Second, each statement's CP is propagated to definitions of privatizable variables used in that statement to ensure necessary values are available. Finally, to ensure that each processor will reach each statement whose instances it must execute (in the presence of arbitrary conditional control flow), a propagation phase is used to establish the following invariants: (1) each control flow statement is assigned the union of the CPs of the statements that are control dependent on them, and (2) each branch target is assigned a computation partitioning that is a superset of the partitionings assigned to each of its control flow predecessors. These invariants ensure that all processors that need to reach a statement can and do.

3.3 An Integer Set Framework for Program Analysis and Optimization

The previous sections described the general computation partitioning (CP) framework in the dHPF compiler, and its use in implementing a number of powerful computation partitioning optimizations. One of the major challenges in supporting such a general framework is the analysis required to support communication analysis and communication optimization, which must take CPs into account explicitly. More generally, the analysis and code generation techniques used to perform communication optimizations are a key determinant of the performance of the generated parallel code.

Most research and commercial data-parallel compilers to date [LC91, BCZ92, KM91, SOG94, HKT93, GB92, GMS+95, HBB+95, BMN+95, GKHS96, vDSP96, ZBG88] perform communication analysis and code generation by considering specific combinations of the form of references, data layouts and computation partitionings. Such case-based analysis has been the principal implementation technique for data-parallel compilation systems because it provides a practical and conceptually simple strategy for developing compilers, and it can be relied on to yield excellent performance for common cases. The case-based approach, however, provides poor performance for cases that have not been explicitly considered. Perhaps more significantly, such case-based compilers require a relatively high development cost for each new optimization because the analysis and code generation for each case is handled separately. This makes it difficult to achieve wide coverage with optimizations to offer consistently high performance.

A few researchers have used analysis techniques based on linear inequalities, which enable a more general and flexible approach than case-based analysis for implementing data parallel languages [ACIK93, BCG+95, AL93]. This work has focused on computing local communication and iteration sets, and performing code generation from these sets using Fourier-Motzkin elimination (FME) [AI91]. Amarasinghe and Lam [AL93] also describe how inequalities and FME can support array dataflow analysis and a few specific communication optimizations. A limitation of the linear inequality representations used by these groups is that they cannot represent general non-convex sets. These representations therefore preclude optimizations such as coalescing communication for arbitrary affine references [AL93], non-local index-set splitting [KM91], or the use of a general computation partitioning model such as that used in our work.

Recently, Pugh et al. have reported important progress in developing efficient algorithms for manipulating and generating code from general non-convex sets [Pug92, KPR95]. This provides a potentially important new tool for developing parallelizing compilers, but leaves open two key questions: Is it practical to use these techniques for real programs? And what challenges must be addressed in formulating and implementing the key analysis and optimizations problems using such an approach?

The Rice dHPF compiler is based on an abstract integer set framework that expresses data parallel program analyses and optimizations in terms of operations on symbolic integer sets. Using this framework, we have devised and implemented simple, concise, yet general, formulations of the major partitioning and communication analyses as well as a number of important optimizations. Because of the simplicity of these formulations, it has been possible to implement a comprehensive collection of advanced optimizations in dHPF. In most cases, our implementations of the optimizations are more general than in previous compilers. Furthermore, all these analyses and optimizations fully support the general CP framework used in dHPF, which is also much more general than that used in previous compilers as described earlier.

The key data-parallel program analyses and optimizations that we have implemented using integer sets include:

- communication analysis for a general computation partitioning (CP) model;
- communication vectorization for arbitrary regular communication patterns;
- message coalescing for arbitrary affine references to an array;
- a powerful class of loop-splitting transformations that support several optimizations, including overlapping communication with computation *within* a single loop-nest, and reducing the overhead of accessing buffered non-local data; and
- a combined compile-time/run-time algorithm to reduce explicit data copies for a message.

All of these analyses and optimizations have been implemented in the dHPF compiler. Of these, the CP model, the loop-splitting based optimizations, the analysis for buffering non-local data, and message coalescing are all more general than in previous compilers.

Another major advance in the dHPF compiler is related to a fundamental limitation of a general integer-set based representation, namely that set constraints containing products of integer variables yield problems that are undecidable [KMP+96]. The primary source of such symbolic product terms is the HPF `distribute` directive with unknown processor counts or block sizes. We describe a natural extension to our framework based on a virtual processor model that supports these symbolic terms without requiring any changes to the set formulations for the above optimizations. A key component of this extension is an integer-set algorithm and associated code-generation strategy that eliminates or reduces runtime checks by restricting the computation and communication to the *active* virtual processors. This extension makes it possible to compile HPF programs for an

data_k:	the index set of an array of rank $k, k \geq 0$
loop_k:	the iteration space of a loop nest of depth $k, k \geq 0$
proc_k:	the processor index space in a processor array of rank $k, k \geq 1$
Layout:	$\text{proc}_n \rightarrow \text{data}_k$: $\{[\underline{p}] \rightarrow [\underline{a}]$: array element $\underline{a} \in \text{data}_k$ is allocated to processor $\underline{p} \in \text{proc}_n\}$
RefMap:	$\text{loop}_k \rightarrow \text{data}_n$: $\{[\underline{i}] \rightarrow [\underline{a}]$: array element $\underline{a} \in \text{data}_k$ is referenced in iteration $\underline{i} \in \text{loop}_k\}$
CPMap:	$\text{proc}_n \rightarrow \text{loop}_k$: $\{[\underline{p}] \rightarrow [\underline{i}]$: statement instance $\underline{i} \in \text{loop}_k$ is assigned to processor $\underline{p} \in \text{proc}_n\}$

Figure 1: Primitive sets and mappings for compiling data-parallel programs.

```
real A(0:99,100), B(100,100)
processors P(4)
template T(100,100)
align A(i,j) with T(i+1,j)
align B(i,j) with T(*,i)
distribute t(*,block) onto P

read(*), N
do i = 1, N
   do j = 2, N+1
               !on_homeB(j-1,i)
      A(i,j) = B(j-1,i)
   enddo
enddo
```

symbolic N

$$\begin{aligned}
\text{proc} &= \{[p] : 1 \leq p \leq 4\} \\
\text{Align}_A &= \{[a_1, a_1] \rightarrow [t_1, t_2] : t_1 = a_1 + 1 \wedge t_2 = a_2\} \\
\text{Align}_B &= \{[b_1, b_2] \rightarrow [t_1, t_2] : t_2 = b_1\} \\
\text{Dist}_T &= \{[t_1, t_2] \rightarrow [p] : 25p + 1 \leq t_2 \leq 25(p+1) \wedge 0 \leq p \leq 3\} \\
\text{Layout}_A &= \text{Dist}_T^{-1} \circ \text{Align}_A^{-1} \\
&= \{[p] \rightarrow [a_1, a_2] : max(25p + 1, 1) \leq a_2 \leq min(25p + 25, 100) \wedge \\
&\qquad\qquad 0 \leq a_1 \leq 99\} \\
\text{Layout}_B &= \text{Dist}_T^{-1} \circ \text{Align}_B^{-1} \\
&= \{[p] \rightarrow [b_1, b_2] : max(25p + 1, 1) \leq b_1 \leq min(25p + 25, 100) \wedge \\
&\qquad\qquad 1 \leq b_2 \leq 100\} \\
\text{loop} &= \{[l_1, l_2] : 1 \leq l_1 \leq N \wedge 2 \leq l_2 \leq N + 1\} \\
\text{CPRef} &= \{[l_1, l_2] \rightarrow [b_1, b_2] : b_2 = l_1 \wedge b_1 = l_2 - 1\} \\
\text{CPMap} &= \text{Layout}_B \circ \text{CPRef}^{-1} \textstyle\bigcap_{\text{range}} \text{loop} \\
&= \{[p] \rightarrow [l_1, l_2] : 1 \leq l_1 \leq min(N, 100) \wedge \\
&\qquad max(2, 25p + 2) \leq l_2 \leq min(N + 1, 101, 25p + 26)\}
\end{aligned}$$

Figure 2: Construction of primitive sets and mappings for an example program. (Align_A, Align_B, and Dist_T also include constraints for the array and template ranges, but these have been omitted here for brevity.)

unspecified number of processors, and we present experimental results to show that there is little or no difference in compile-time for a symbolic than for a constant number of processors, even on fairly large and complex codes such as the NAS SP application benchmark.

In the following three sections, we describe the set framework, some of the optimizations implemented using the framework, and the framework extension for a symbolic number of processors. Section 3.6.1 presents experimental results showing that the set representation is not a dominant factor in compile times on both small and large codes, as well as results evaluating the performance of the compiler generated parallel code.

3.3.1 Description of the Set Framework for Data-Parallel Compilation

An integer k-tuple is a point in $\mathcal{Z}^k$; a tuple space of rank k is a subset of $\mathcal{Z}^k$. Any compiler for a data-parallel language based on data distributions, such as HPF, operates primarily on three types of tuple spaces, and the three pairwise mappings between these tuple spaces ([AL93, HKT92, KM91]). These are shown in figure 1.[2] Scalar quantities such as a "data set" for a scalar, or the "iteration set" for a statement not enclosed in any loop are handled uniformly as tuples of rank zero. Hereafter, the terms "array" and "iterations of a statement" imply scalars and outermost statements as well.

[2]We use names with lower-case initial letters for tuple sets and upper-case letters for mappings respectively.

Inputs:

$\mathrm{Refs_{read}}, \mathrm{Refs_{write}}$	:	sets of read and write references in a single logical communication event
$\mathrm{RefMap_r}$	:	map representing reference r, $\forall \mathrm{r} \in \mathrm{Refs_{read}} \cup \mathrm{Refs_{write}}$
$\mathrm{CPMap_r}$	:	computation partitioning map for reference r
$\mathrm{Layout_A}$	:	layout of the common referenced array, denoted A
$\mathrm{V_r}$	:	loop-level of innermost loop enclosing communication for r, after vectorization; $J_1 \ldots J_{V_r}$ are index variables of enclosing loops
$\underline{\mathrm{m}}$	:	processor index vector for the representative processor m or *myid*

Algorithm:

$$
\begin{array}{lrcl}
(1) & \mathrm{CPMap_r^V} & = & CPMap_r \bigcap_{range} \{[j_1, \ldots, j_n] : j_1 = J_1 \wedge \ldots j_{V_r} = J_{V_r} \wedge -\infty < j_{V_r+1} < \infty \wedge \ldots\} \\
(2) & \mathrm{DataAccessed_t} & = & \bigcup_{\mathrm{r \in Refs_t}} \mathrm{CPMap_r^V} \circ \mathrm{RefMap_r}, \quad (\text{hereafter, } t \in \{read, write\}) \\
(3) & \mathrm{NLDataAccessed_t} & = & \{[\underline{\mathrm{p}}] \rightarrow [\underline{\mathrm{a}}] : \text{off-processor array elements } \underline{\mathrm{a}} \text{ referenced by processor } \underline{\mathrm{p}}\} \\
 & & = & \begin{cases} \mathrm{DataAccessed_t} - \mathrm{Layout_A} & \text{if } t = read \\ \mathrm{DataAccessed_t} \cap \mathrm{Layout_A} & \text{if } t = write \end{cases} \\
 & \mathrm{nlDataSet_t(m)} & = & \mathrm{NLDataAccessed_t(\{\underline{m}\})} \\
(4) & \mathrm{NLCommMap_t(m)} & = & \{[\underline{\mathrm{p}}] \rightarrow [\underline{\mathrm{a}}] : \text{off-processor elements referenced by proc. } m \text{ and owned by proc. } \underline{\mathrm{p}}\} \\
 & & = & \mathrm{Layout_A} \bigcap_{\mathrm{range}} \mathrm{nlDataSet_t(m))} \\
(5) & \mathrm{LocalCommMap_t(m)} & = & \{[\underline{\mathrm{p}}] \rightarrow [\underline{\mathrm{a}}] : \text{array elements owned by proc. } m \text{ to be communicated with proc. } \underline{\mathrm{p}}\} \\
 & & = & \mathrm{DataAccessed_t} \bigcap_{\mathrm{range}} \mathrm{Layout_A(\{\underline{m}_A\})} \\
(6) & \mathrm{SendCommMap(m)} & = & \mathrm{LocalCommMap_{read}(m)} \bigcup \mathrm{NLCommMap_{write}(m)} \\
(7) & \mathrm{RecvCommMap(m)} & = & \mathrm{NLCommMap_{read}(m)} \bigcup \mathrm{LocalCommMap_{write}(m)}
\end{array}
$$

Figure 3: Equations for computing communication sets

The sets loop and proc and the mappings Layout and RefMap are computable directly from the source program and form the primary inputs for further analyses. Figure 2 illustrates these primitive sets and mappings for an HPF code fragment. Layout is computed from Align, which represents the alignment of an array with a template, and Dist, which represents the distribution of a template on a physical processor array. The iteration set, loop, follows directly from the loop bounds. The `on_home` CP notation and construction of CPMap are described in the following section.

To implement analysis and optimization in dHPF as operations on these symbolic sets and mappings, we require an integer set package that supports operations including intersection, union, difference, domain, range, composition, and projection. For this purpose, we use the Omega Library developed by Pugh et al at the University of Maryland [KMP+96]. Omega supports representation and manipulation of (potentially non-convex) integer sets described by Presburger formulae, using algorithms based on Fourier-Motzkin Elimination (FME) [Pug92]. This approach has both advantages and disadvantages compared to simpler set representations such as extended Rectangular Sections [HKT93] or Data Access Descriptors [Bal90]. In Section 3.3.3, we discuss the tradeoffs that arise and describe how we accommodate one of the key limitations of FME.

3.3.2 Optimizations using the Integer Set Framework

The integer set framework described in the previous section provides the basis for sophisticated program optimizations in the dHPF compiler. Our set framework enables us to analyze and optimize programs with complex computation partitionings, as described in Section 3.2. In this section, we describe how we use our integer set framework for analysis and optimization.

Implementing Explicit Communication

For message-passing systems, data-parallel compilers must compute the data to be exchanged between processors, and generate code to pack, communicate, unpack, and reference the non-local data. *Message vectorization* is a fundamental optimization for such systems which reduces communication frequency by hoisting communication for a reference out of one or more enclosing loops. To vectorize communication, the compiler must compute the set of data to send between each pair of processors; such communication sets depend on the reference, layout, and computation partitioning. *Message coalescing* combines messages for multiple references to eliminate redundant communication and further reduce the number of messages. Coalescing requires unioning communication sets and can require sophisticated techniques to enable efficient access to buffered non-local data.

Early phases in dHPF identify potentially non-local references that might access off-processor data, compute where to place communication for each reference (using dependence and optionally dataflow analysis to vectorize communication), and which sets of references can have their communication coalesced. dHPF assumes that all data is communicated to and from its owner(s) only, as defined by the data layout directives. A read reference is non-local if the location is not owned by the processor executing the read. A write reference is non-local if the location is owned by one or more processors besides the processor executing the write. (These definitions are equivalent if data is not replicated.) We refer to the sequence of messages required for a set of coalesced references as a single *logical communication event.*

Given sets of coalesced references and the placement level of communication, we compute the communication sets for each logical communication event using the inputs and set equations shown in Figure 3. The goal of these equations is to compute two maps, SendCommMap(m) and RecvCommMap(m), for the representative processor $m = \texttt{myid}$. The maps respectively specify the data that processor m must send to each other processor $\underline{p}$ and the data that processor m must receive from each other processor $\underline{p}$.

The key operations are as follows. Steps 1 and 2 compute the two maps $\mathrm{DataAccessed_t}$, $t \in \{read, write\}$, which specify the entire set of data accessed by each processor $\underline{p}$, via all read and write references in all iterations of the loops out of which communication has been vectorized. Then (step 3), the *non-local* data accessed by read references is the difference of $\mathrm{DataAccessed_t}$ and the local data owned by each processor, $\mathrm{Layout_A}$. The non-local data accessed by write references is the intersection of $\mathrm{DataAccessed_t}$ and the data owned by any other processors. Note that the read and write equations are equivalent for the common case that each array element is owned by a single processor, i.e., where the data layout is not replicated.[3] We convert this map to a set $\mathrm{nlDataSet_t(m)}$ specifying the non-local data accessed by the fixed processor m.

We then compute two maps describing the non-local and local data (w.r.t. to the fixed processor m) that must be communicated with each other processor $\underline{p}$ (steps 4,5). Restricting the range of $\mathrm{Layout_A}$ to the non-local data set, $\mathrm{nlDataSet_t(m)}$, gives the non-local data referenced by processor

[3] In this case, in fact, we can skip step (3) and simply use $NLDataAccessed_t = DataAccessed_t$, because steps (4) and (5) will restrict communication to non-local data. During code generation, we ensure that a processor does not communicate with itself.

m and owned by each other processor $\underline{p}$. Restricting the range of DataAccessed$_t$ to the local section owned by m gives the local data owned by m and accessed by each other processor $\underline{p}$.

Finally, from LocalCommMap$_t$ and NLCommMap$_t$, $t \in \{read, write\}$, we compute the data to send to and receive from each processor (steps 6,7). Later, we generate code from these maps to pack and unpack the data at the sending and receiving ends.

The equations presented here unify the handling of both communication vectorization and coalescing for arbitrary references and communication patterns. This abstract formulation of static communication analysis has greatly simplified the core of the dHPF compiler and enabled efficient handling of general classes of computation partitionings and affine references.

Recognizing in-place communication

Common MPI implementations permit data to be sent or received "in-place" (avoiding an explicit data copy) when the address range of the data is contiguous. To increase the likelihood that communication can be performed in-place, we develop a combined compile-time/run-time algorithm for recognizing contiguous data based on the capability of generating code from integer sets.

FORTRAN arrays are stored in column-major order. Accordingly, a *rectangular* communication set C for data in an array `A` with n dimensions represents contiguous data if there exists a k such that for the dimensions $1 \leq i < k$, the set spans the full range of array dimension i, along dimension k the set has a contiguous index range, and in the low-order dimensions $k+1 \leq j \leq n$, the set contains a single index value. Let A represent the local index set of the array, and define $S_{<i>}$ to be the projection (i.e., range) of set S in dimension i, $1 \leq i \leq \text{rank}(S)$. Then the above condition can be formalized as:

$$\exists k \ s.t.\ 1 \leq k \leq n \ \wedge \ \bigwedge_{i=1}^{i=k}(C_{<i>} = A_{<i>}) \ \wedge$$
$$\text{IsConvex}(C_{<k>}) \ \wedge \ \bigwedge_{i=k+1}^{i=n}\text{IsSingleton}(C_{<i>})$$

To permit runtime evaluation when necessary, we reduce each of the tests to a satisfiability question for which we can synthesize an equivalent conditional to be tested at run time (if it cannot be proved at compile-time). The predicate IsConvex(S) reduces to testing if the set ConvexHull$(S) - S$ is empty. The predicate IsSingleton(S) also reduces to a satisfiability test, but we omit the details here.

To avoid evaluating O(n^2) predicates at compile-time, we use a single scan of the dimensions (leftmost first) to find the first dimension k for which $C_{<k>} \neq A_{<k>}$, and check the predicates for $k \ldots n$. If in-place communication cannot be proven at compile time, we synthesize code from the unproven predicates to repeat this scan and check at runtime, when it can be done precisely by evaluating at most $n+2$ predicates. In general, this test can be performed much faster than packing a buffer of modest size. This approach, based on explicit integer sets, enables us to exploit in-place communication for arbitrary communication sets, independent of data layouts and communication patterns.

There are currently two limitations of our implementation of this analysis in the dHPF compiler. First, we apply the compile-time test for in-place communication only to communication sets with only a single conjunct. Our compiler support can be generalized straightforwardly to handle disjoint disjunctions as well when the satisfiability conditions on all conjuncts are mutually exclusive. Second, the code generation for runtime evaluation of these predicates is currently incomplete.

Loop Splitting

Loop splitting (or non-local index set splitting) is a powerful but complex transformation that has been proposed to ameliorate two types of communication overhead: the cost of referencing buffered non-local data, and the latency of communication [KM91]. Both techniques involve splitting a loop to separate the iterations that access only local data from those that may access non-local data. First, buffer access costs arise when local and non-local data are stored separately, and the correct location must be chosen with a runtime check on each reference. After splitting local and non-local iterations, no checks are needed for references in the local iterations. Second, the latency of communication can be (partly) hidden by splitting because communication required for non-local iterations can be overlapped with computation of the local iterations.

The only implementation of this transformation we know of is in Kali [KM91], where the authors used set equations to explain the optimization but used case-based analysis to derive the iteration sets for special cases limited to one-dimensional distributions. This approach is only practical for a small number of special cases.

We extend the equations in [KM91] to apply to arbitrary sets of references, and any CP in our CP model. We first describe loop-splitting for communication overlap, because it subsumes splitting for buffer access. We apply loop-splitting to any perfect loop-nest (not necessarily innermost) containing at least one partitioned loop and having no dependences that prevent iteration reordering. Since, in dHPF, write references may be non-local, we split the set of iterations of such a loop nest into four sections: those that access only local data (localIters), and those that only read, only write, or read and write non-local data (nlROIters, nlWOIters and nlRWIters respectively). These sets are computed as shown in Figure 4(a) for a loop-nest containing one or more potentially non-local references. (The equations shown are applied separately to each statement group in the loop nest.) The key steps are to compute $\mathrm{localDataAccessed}_r$ (analogous to computing $\mathrm{nlDataSet}_t$ in Figure 3), and then $\mathrm{localIters}_r$ which are the iterations in which reference r accesses local data. The desired four sets are then computed by grouping $\mathrm{localIters}_r$ by read and write references.

We schedule the communication and computation for this loop nest in the sequence shown in Figure 4(b). When NLRW is non-empty, we can overlap either read or write latency, but not both; a simple heuristic could be used to choose between the two. The sequence in the figure overlaps read latency with NLWOIters and LocalIters. When NLRW is empty, however, the latency for writes as well as reads can be overlapped with LocalIters by placing the SEND for non-local writes immediately after NLWOIters.

This form of splitting subsumes splitting for non-local buffer access. References in the local iterations do not need buffer-access checks, and a reference r in a non-local loop section (e.g., NLROIters) also does not need such checks if $\mathrm{nlROIters} \subset \mathrm{nlIters}_r = \mathrm{cpIterSet} - \mathrm{localIters}_r$, because the reference will access only non-local data in these iterations.

Code generation for loop splitting subsumes the operation of partitioning the loop by reducing the loop bounds, since each of the four loop sections is a subset of cpIterSet for the statement group. The code generation is performed as part of the hierarchical code generation framework for computation partitioning described briefly in Section 3.2 [AMC98a].

3.3.3 Extensions for Symbolic Distribution Parameters

A variety of set representations can be used to implement the set framework, with a wide range of expressiveness and efficiency. We observe that the primary benefit of using the integer set approach is that it enables simple but rigorous formulations of the key data-parallel optimization problems. This benefit is somewhat independent of the generality of the underlying set representation. The Omega library provides a powerful integer set representation based on FME, and the library has been invaluable in developing and prototyping the set-based formulations of the optimizations. Nevertheless, FME has two potential disadvantages which we address as follows.

First, algorithms based on FME have poor worst-case performance. A goal of our research has

Inputs:

CPMap	:	Common CP map for each statement in given statement group SG
$\mathrm{Refs}_{\mathrm{read}}, \mathrm{Refs}_{\mathrm{write}}$	:	non-local read and write references in SG
RefMap_r	:	map representing reference r, $\forall r \in \mathrm{Refs}_{\mathrm{read}} \cup \mathrm{Refs}_{\mathrm{write}}$
Layout_A	:	layout of the common referenced array, denoted A
$\underline{m}_{CP}, \underline{m}_A$	:	processor index vectors as in Figure 3

Algorithm:

$$
\begin{aligned}
\mathrm{cpIterSet} &= \mathrm{CPMap}(\{\mathbf{m}\}) \\
\mathrm{dataAccessed}_r &= \mathrm{RefMap}_r(\mathrm{cpIterSet}) \\
\mathrm{localDataAccessed}_r &= \begin{cases} \mathrm{dataAccessed}_r \cap \mathrm{Layout}_A(\{\underline{m}_A\}) & \text{if } t = \text{read} \\ \mathrm{dataAccessed}_r - \mathrm{Layout}_A(\neg\{\underline{m}_A\}) & \text{if } t = \text{write} \end{cases} \\
\mathrm{localIters}_r &= \mathrm{Ref}_r^{-1}(\mathrm{localDataAccessed}_r) \\
\mathrm{nlReadIters} &= \mathrm{cpIterSet} - \bigcap_{r \in \mathrm{Refs}_{\mathrm{read}}} \mathrm{localIters}_r \\
\mathrm{nlWriteIters} &= \mathrm{cpIterSet} - \bigcap_{r \in \mathrm{Refs}_{\mathrm{write}}} \mathrm{localIters}_r \\
\mathrm{localIters} &= \mathrm{cpIterSet} \cap \bigcap_{r} \mathrm{localIters}_r \\
\mathrm{nlRWIters} &= \mathrm{nlReadIters} \cap \mathrm{nlWriteIters} \\
\mathrm{nlROIters} &= \mathrm{nlReadIters} - \mathrm{nlWriteIters} \\
\mathrm{nlWOIters} &= \mathrm{nlWriteIters} - \mathrm{nlReadIters}
\end{aligned}
$$

(a) Computing local/nonlocal iteration sets

```
SEND data for non-local reads
execute NLWOIters
execute LocalIters
RECV data for non-local reads
execute NLROIters ⋃ NLRWIters
SEND data for non-local writes
RECV data for non-local writes
```

(b) Scheduling loop iterations

Figure 4: Loop splitting to overlap communication and computation.

been to determine whether or not this general approach is practical for commercial HPF compilers. There is some previous evidence that poor cases may be unlikely to occur in practice [Pug92]. Also, changing the formulation of a problem can be extremely effective in avoiding complex sets, and has improved running times by more than an order of magnitude in some cases. In practice, we have not found the cost of the set framework to be a problem so far, as shown in Section 3.6.1. Nevertheless, if the approach still proves impractical, we can use one of two alternatives (without sacrificing the benefits of the simple equational formulations). We can use a simpler set representation such as Data Access Descriptors [Bal90], or we can use competitive algorithms that limit the time spent on any single optimization or code generation problem. Both approaches would fall back on runtime techniques (such as inspector-executor [SCMB90]) which are required in any case for irregular problems.

A second, fundamental limitation of Omega is that it does not permit symbolic coefficients in affine constraints, because multiplication of integer variables renders the underlying class of integer sets undecidable [Coo72]. Symbolic multiplication is required to represent a symbolic stride, k, for example, $\{[i] : 1 \le i \le N \wedge \exists \alpha \ s.t. \ i = k\alpha + 1\}$. In compiling an HPF program, symbolic strides arise for any type of HPF distribution when the number of processors is unknown at compile time, for the *cyclic*(k) distribution with unknown k, and for loops with unknown strides. We have extended our framework to accommodate the limitation on symbolic number of processors and k, as described below. Loops with unknown strides are not supported by our framework, and would have to fall back on more expensive run-time techniques such as a finite-state-machine approach

Inputs: Same as in Figure 3
Algorithm:

$$\begin{aligned}
\text{busyVPSet}_t &= \bigcup_{r \in Refs_t} Domain(CPMap_r), \quad \text{(hereafter, } t \in \{read, write\}) \\
\text{allNLDataSet}_t &= \text{NLDataAccessed}_t(\text{busyVPSet}_t) \\
\text{vpsThatOwnNLData}_t &= \text{Layout}_{A_r}^{-1}(\text{allNLDataSet}_t) \\
\text{vpsThatAccessNLData}_t &= \text{Domain}(\text{NLDataAccessed}_t) \\
\text{activeSendVPSet} &= \text{vpsThatOwnNLData}_{\text{read}} \bigcup \text{vpsThatAccessNLData}_{\text{write}} \\
\text{activeRecvVPSet} &= \text{vpsThatAccessNLData}_{\text{read}} \bigcup \text{vpsThatOwnNLData}_{\text{write}}
\end{aligned}$$

(a) Equations for computing the active virtual processors

```
real A(1:100)
processors PA(P1,P2)
template T(100,100)
align A(i,j) with T(i,j)
distribute t(cyclic,cyclic) onto PA
...
do i = PIVOT+1, 100
  do j = PIVOT+1, 100
    !on_home{ A(i,j) }
    A(i,j) = ··· + A(PIVOT,j)
  enddo
enddo
```

(b) Gauss parallel loop in HPF

$$\begin{aligned}
\text{vpArray} &= \{[v_1, v_2] : 1 \le v_1, v_2 \le 100\} \\
\text{loop} &= \{[i,j] : \text{PIVOT} + 1 \le i, j \le 100\} \\
\text{CPMap} &= \{[v_1, v_2] \to [i,j] : i = v_1 \wedge j = v_2 \wedge \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{RefMap}_{\text{read}} &= \{[i,j] \to [\text{PIVOT}, j]\} \\
\text{RefMap}_{\text{write}} &= \emptyset \text{(no non-local writes)} \\
\text{busyVPSet}_{\text{read}} &= \{[v_1, v_2] : \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{NLDataAccessed}_{\text{read}} &= \{[v_1, v_2] \to [\text{PIVOT}, v_2] : \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{activeSendVPset} &= \{[v_1, v_2] : v_1 = \text{PIVOT} \wedge \text{PIVOT} < v_1, v_2 \le 100\} \\
\text{activeRecvVPset} &= \text{busyVPSet}_{\text{read}}
\end{aligned}$$

(c) Active virtual processors in Gauss loop

Figure 5: Active virtual processors for computing, sending and receiving

for computing communication and iteration sets (for example, [KNS95]).

3.3.4 An Optimized Virtual Processor Model

To circumvent the limitation on symbolic data distribution parameters, we use a standard technique that avoids representing these parameters explicitly within the set framework, and instead incorporates them directly during code generation [AL93, GKHS96]. We refine this technique to provide significantly simpler code for *block* distributions (by far the most common in practice). We also add an additional optimization to to eliminate or reduce the runtime overhead in the resulting code. Gupta et al. [GKHS96] apply a similar strategy but their approach was based on detailed analysis of specific cases. Instead, we describe a general integer-set-based algorithm to perform this optimization. The VP model and optimization are as follows.

To avoid representing the distribution explicitly, we replace each physical processor array in our equations by a virtual processor (VP) array corresponding to using template indices (i.e., ignoring the distribute directive) in dimensions where the block size or number of processors is unknown, but using one virtual processor index per physical processor index in all other dimensions. (We do not require template sizes to be known constants.) We make this replacement simply by constructing the Layout mapping as a map from VP indices to data elements. All the analyses described in the previous sections then operate unchanged on this virtual processor domain. During code generation for each specific problem (e.g., generating a partitioned loop), we add extra enclosing loops that

```
do v = vlb, vub
   ! pack data for v
   ! send data to v
enddo
```

(a) Initial SEND code from Domain(SendDataMap(m))

```
do firstVP = vlb, min(vub, vlb+P)
   pid = (firstVP-tlb) % P + tlb
   do v = firstVP, vp2, P
      ! pack data for v
   enddo
   ! send data to pid
enddo
```

(b) Separating physical and virtual processors

```
myVLB = ((activeSendLB-tLB)/P)*P + m + tLB
if (myVLB .lt. activeSendLB) myVLB = myVLB + P
do firstVP = vlb, min(vub, vlb+P)
   pid = (firstVP-tlb) % P + tlb
   do myVP = myVLB, activeSendUB, P
      do v1 = firstVP, vub, P
         ! pack data for v1
      enddo
   enddo
   ! send data to pid
enddo
```

(c) Adding active virtual processor loop (final SEND code)

Figure 6: Code generation for SEND with optimized virtual processor model

enumerate the VPs that are owned by the physical processor `myid`. Also, when generating code for communication, we must aggregate the messages to all the VPs belonging to the same physical processor. These code generation steps are described in Section 3.3.5.

The refinement we use for *block* distributions is as follows. Consider a template dimension of extent N that is *block*-distributed on a processor array dimension of extent P, and let $B = \lceil N/P \rceil$ be the block size. The precise representation of this distribution is $\{[t] \rightarrow [p] : Bp + 1 \leq t \leq Bp + B \wedge 1 \leq t \leq N \wedge 1 \leq p \leq P\}$. In this case, only the product term Bp is not directly representable. We compute a distribution onto virtual processors as follows: $\{[t] \rightarrow [v] : v \leq t \leq v + B - 1 \wedge 1 \leq t \leq N \wedge 1 \leq v \leq N\}$. Intuitively, this assumes that the virtual processor v "owns" template elements $[v, v + B - 1]$. Each physical processor p is mapped to a unique virtual processor $v = Bp + 1$. However, any other value of v represents a fictitious virtual processor not mapped to any physical processor. The special physical processor id $\underline{m}$ used in Figures 3 and 4 is replaced by its virtual processor id $v_{m_i} = Bm_i + 1$. Therefore, the generated SPMD code is automatically parameterized by $\underline{v_m}$ instead of $\underline{m}$.

There are two key advantages in using this refinement for the *block* distribution, and both are due to the property that there is a single virtual processor per physical processor. First, this property implies that no additional virtual processor loops are required. In fact, $\underline{v_m}$ always represents a true physical processor (*myid*), and therefore a *block* distribution does not require any additional changes to the computational loops in the generated SPMD code (even with transformations such as non-local index set splitting). Second, the above property implies that the communication sets capture the entire section that must be communicated to each physical processor (for each *block* dimension). This enables some optimizations and simplifies code generation. For example, if all array dimensions are *block*-distributed, the equations for recognizing in-place communication can be applied to the communication set to determine whether in-place communication is feasible. For code generation, the only additional step required is to ensure that communication is not attempted with a fictitious virtual processor. This step is described in Section 3.3.5.

For *cyclic* and *cyclic*(k) distributions, there are multiple virtual processors for each physical processor. However, not all virtual processors owned by a physical processor are necessarily "active" in any particular operation (a partitioned computation, sending data, or receiving data). An additional optimization step can be used to eliminate or reduce runtime overhead by restricting the virtual processor loop to the set of VPs owned by processor *myid* that are actually active. Since such a set for a single physical processor would not be directly representable, in general, we do this in two steps. We first use the integer-set equations shown in Figure 5(a) to compute the set of active VPs *across all processors* for the problems of interest. Then, we generate a loop nest from this set and explicitly rewrite the loop bounds to restrict it to the active VPs owned by processor *myid*.

First, the set of active VPs for any partitioned computation, denoted busyVPSet, is simply the domain of CPMap. (As in Figure 4, this must be applied for each statement group in a loop nest.) Second, for each logical communication event, the active VPs that must send or receive data can be computed directly from NLDataAccessed_t, the map from processors to non-local data referenced by each processor ($t \in \{read, write\}$). This map is already computed for communication generation (Figure 3). The map is first used to compute the sets of all virtual processors that own non-local data and all those that access non-local data ($\text{vpsThatOwnNLData}_t$ and $\text{vpsThatAccessNLData}_t$). These in turn directly provide the virtual processors that must be active in sending or receiving data.

The results of these equations are illustrated for the Gaussian Elimination example in Figure 5(b,c), where the reference to the pivot row, $A(PIVOT, j)$, requires off-processor data. The busyVPSet reflects that only VPs corresponding to the lower, right portion of the matrix A are active. activeSendVPSet and activeRecvVPSet indicate that only VPs owning elements in the pivot row (PIVOT) must send any data, but all VPs active in the computation (busyVPSet) must receive non-local data. In practice, we can generate code so that only one VP per physical processor will receive each such element.

3.3.5 Code Generation using Virtual Processors

Given the active virtual processor sets computed above, a few conceptually simple steps are required to generate final SPMD code.

For a computational loop nest, if the CPs of all statement groups are described via *block*-distributed arrays, no additional steps are required as explained earlier. If any array dimension used in a given CP has a *cyclic* or *cyclic*(k) distribution, we must enclose the loop nest with one extra loop for each such dimension. We generate these loops as follows. We first generate a loop nest to enumerate the elements of the set busyVPSet computed from the CP. For each loop in the nest, we then directly modify its bounds and stride to restrict the loop to the active VPs owned by processor m. For example, for the i dimension in Figure 5, the final VP loop would have $lb = (PIVOT/P) * P + m + 1 + (m < PIVOT\%P)?P : 0$, $ub = 100$, and $step = P1$. If non-local index-set splitting is applied to the loop nest, the same virtual processor loop nest would have to be wrapped around each loop nest section.

For communication code, we describe the steps required for the sending side; the receiving code is similar. The steps are illustrated in Figure 6, assuming a 1D processor array with P processors, a template with lower bound tLB, and a *cyclic* distribution. First, we generate a loop nest to enumerate Domain(SendDataMap(m)), viz., the virtual processors to which processor m must send data, and insert code to pack data for v by enumerating Range(SendDataMap(m)) (part (a) of the figure). Then, we rewrite this loop to separately enumerate the physical processors, and the virtual processors for each physical processor (part (b)). Finally, we enumerate the active sending processors owned by m, by applying the procedure described above (for busyVPSet) to the set activeSendVPSet. This loop nest is wrapped around the inner virtual processor (packing) loop, yielding the final send code in part (c). (The *block* distribution case is again much simpler because the two inner virtual processor loops are not required.)

More generally, if the original loop-nest of part (a) had r loops in a (possibly imperfect) loop nest, we generate r physical processor loops with the same nesting structure. If k of these loops correspond to *cyclic* data dimensions, we generate $2k$ virtual processor loops in a single perfect loop nest enclosing the packing code. Finally, we insert one copy of this virtual processor loop nest into each innermost physical processor loop.

The extensions described above enable us to use a very general set representation without unduly restricting the use of symbolic quantities. The additional complexity introduced by these techniques is largely encapsulated in the implementation of the framework, and (we believe) are outweighed by the analysis capabilities and flexibility the integer set framework provides. Measurements described in Section 3.6.1 show that the compile-time cost of the symbolics extension is also not significant.

The extensions for supporting symbolics quantities also play an important role in the techniques for supporting multipartitioning distributions, discussed in the next Section.

3.4 Computation Partitioning Optimizations

The previous section presented the general computation partitioning model supported by the dHPF compiler. In this section we describe three new program optimization strategies used by dHPF that exploit the generality of the partitioning model by using sophisticated computation partitionings to improve parallelism and reduce communication frequency and volume.

3.4.1 Computation Partitioning for Reductions

The dHPF compiler recognizes implicit reduction operations by analysis of loop nests and then determines how to decompose the computation and data across the processors in a fashion that not only parallelizes the application, but also minimizes communication. This is process is describe in detail elsewhere [LMC98]. Here we briefly describe how the general computation partitioning model in dHPF is an enabling technology for generating efficient code for reductions.

There are three steps in parallelizing a reduction operation. We illustrate this process using a SUM reduction consisting of a loop containing the statement `S = S + A(i)` as an example. First, in a reduction preamble, each processor stores the original value of `S` into a temporary variable `T`, and initializes `S` to be zero. Second, in the reduction core, each processor owning a part of array `A` involved in the reduction computes the partial sum its local values into `S`. Finally, in a postamble, the processors accumulate their partial sums using a collective communication operation, and add back the value saved in `T` to get the final sum. We assign a replicated CP to the preamble and postamble, that is, every processor initializes the partial sum and participates in producing the final reduction value using collective communication. The CP for the partial sum computation $S = S + A(i)$ would be ON_HOME $A(i)$ or ON_HOME of one of the references if there are several references on the right hand side.

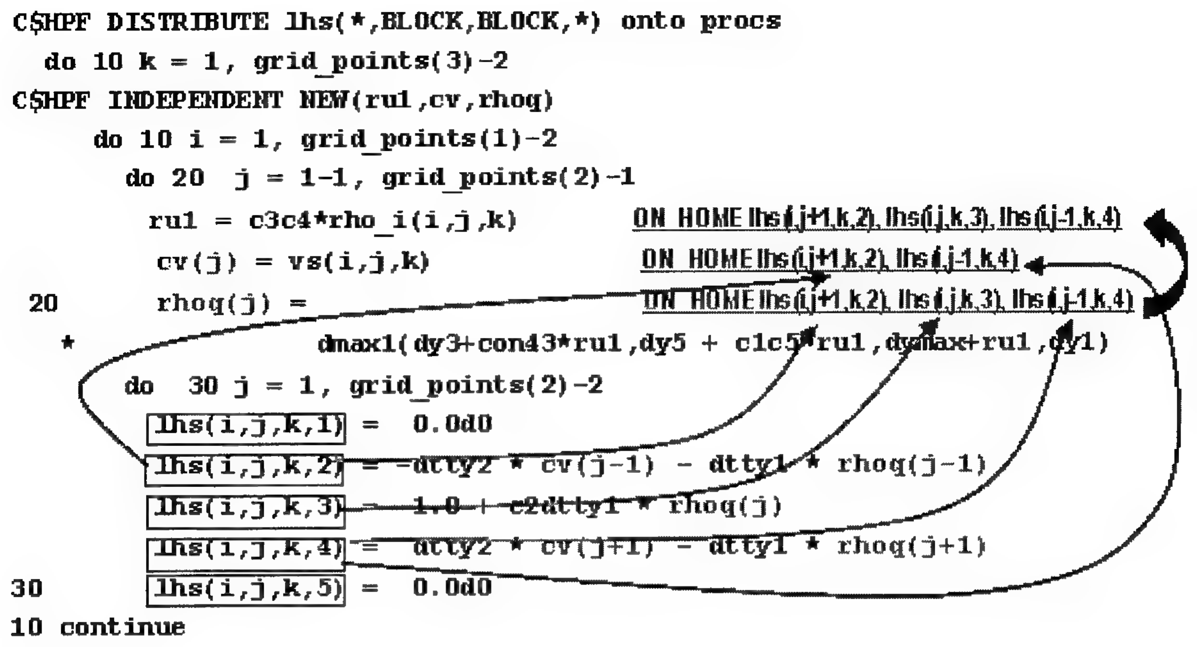
Factorization and Data Locality Suppose we have a reduction statement

$$S = S \oplus A_1(f_1(\vec{i})) \oplus A_2(f_2(\vec{i})) \ldots \oplus A_n(f_n(\vec{i}))$$

in a loop nest where $\vec{i}$ is the vector of enclosing loop indices, $n > 1$, and $\oplus$ is a commutative and associative operator. If some of the arrays $A_1 \ldots A_n$ are distributed differently, some instances of this statement will need to read off-processor data no matter how we specify their CP. For example, suppose we compute the above statement ON_HOME $A_1(f_1(\vec{i}))$. If $A_2(f_2(\vec{i}))$ is not always local to all of the processors owning values referenced by $A_1(f_1(\vec{i}))$, some processors owning elements of $A_1(f_1(\vec{i}))$ will need to read non-local values for $A_2(f_2(\vec{i}))$ to compute the partial sum for their assigned statement instances.

We can eliminate the need to read off-processor data by *factoring* the above reduction statement into a sequence of statements:

$$\begin{array}{ll} S = S \oplus A_1(f_1(\vec{i})), & S = S \oplus A_2(f_2(\vec{i})), \\ \ldots, & S = S \oplus A_n(f_n(\vec{i})) \end{array}$$



```
C$HPF DISTRIBUTE lhs(*,BLOCK,BLOCK,*) onto procs
   do 10 k = 1, grid_points(3)-2
C$HPF INDEPENDENT NEW(ru1,cv,rhoq)
       do 10 i = 1, grid_points(1)-2
          do 20  j = 1-1, grid_points(2)-1
            ru1 = c3c4*rho_i(i,j,k)          ON HOME lhs(i,j+1,k,2), lhs(i,j,k,3), lhs(i,j-1,k,4)
            cv(j) = vs(i,j,k)                 ON HOME lhs(i,j+1,k,2), lhs(i,j-1,k,4)
20          rhoq(j) =                         ON HOME lhs(i,j+1,k,2), lhs(i,j,k,3), lhs(i,j-1,k,4)
    *                 dmax1(dy3+con43*ru1,dy5 + c1c5*ru1,dymax+ru1,dy1)
          do  30 j = 1, grid_points(2)-2
            lhs(i,j,k,1)  =  0.0d0
            lhs(i,j,k,2)  = -dtty2 * cv(j-1) - dtty1 * rhoq(j-1)
            lhs(i,j,k,3)  =  1.0 + c2dtty1 * rhoq(j)
            lhs(i,j,k,4)  =  dtty2 * cv(j+1) - dtty1 * rhoq(j+1)
30          lhs(i,j,k,5)  =  0.0d0
10 continue
```

Figure 7: Computation partitioning for a loop nest from subroutine lhsy of the NAS SP computational fluid dynamics benchmark.

After this transformation, we can compute each reduction statement $S = S \oplus A_k(f_k(\vec{i}))$ $(1 \leq k \leq n)$ ON_HOME $A_k(f_k(\vec{i}))$ without communication. Our compilation model accommodates the factorization process in a natural fashion. Each of the simple statements factored out of a complex statement will be in the same reduction group whose final result will be accumulated by a single collective communication operation in the postamble.

3.4.2 Parallelizing Computations that use Privatizable Arrays

Within complex loop nests in scientific codes, temporary arrays are often used to hold intermediate data values so that they can be reused in the same or subsequent iterations. Typical uses of a temporary array are to save a reference copy of values before they will be overwritten, or to store partial results of a computation that will be used later.

Indiscriminate use of temporary arrays can be an obstacle to effective parallelization of a loop nest because they introduce dependences between different loop iterations. However, in the special case when each element of a temporary array used within a loop iteration is defined within that iteration before its use, and none of the elements defined in the loop are used after the loop, the iterations of the loop nest are fully parallelizable. Such an array is said to be privatizable on the loop. The High Performance Fortran NEW directive provides a convenient syntactic mechanism for programmers to signal a compiler that an array is privatizable.

Even once a compiler knows which arrays are privatizable within a loop nest, it can be challenging to effectively parallelize the loop nest for a distributed-memory machine. For example, consider the loop nest from subroutine lhsy of the NAS 2.3-serial SP benchmark shown in figure 7 with HPF directives added.

In this example, the NEW directive specifies that the cv and rhoq arrays are privatizable within the i loop. The difficulty in this case arises in trying to exploit the potential parallelism of the j loop. When computing the privatizable arrays cv and rhoq, there are two obvious alternatives, both of which are unsatisfactory here. If a complete copy of the privatizable array is maintained on each processor, each processor will needlessly compute all the array elements even though only about 1/P of the values will be used by each processor within the second j loop nest. On the other hand, if the privatizable array is partitioned among processors and each processor computes only the elements it owns, then each processor will have to communicate the boundary values of rhoq and cv to its neighboring processors for use in the second j loop. This would require a large number of small messages between processors, which would also have very high overhead.

In the dHPF compiler, we address these problems by having each processor compute only those elements of the privatizable array that it will actually use [AJMCY98]. When some array elements (such as the boundary elements of cv and rhoq in the example) are needed by multiple processors, the compiler partially replicates the computation of exactly those elements to achieve this goal. This is cost-effective in that it is the minimal amount of replication that completely avoids any communication of the privatizable array in the inner loop. The general CP model in dHPF is crucial for expressing the CPs required by this strategy. For a statement defining a privatizable variable, we assign a CP that is computed from the CPs of each of the statements in which the variable is used. This is done as follows.

For each loop nest, the compiler first chooses CPs for all assignments to non-privatizable variables using a global CP selection algorithm that attempts to minimize communication cost for the loop nest. For the code shown in Figure 7, the algorithm chooses owner-computes CPs for each of the five assignments to array lhs. The CPs for these assignments are represented by the boxed left-hand-side references. We then translate these CPs and apply the translated CPs to the statements that define the privatizable arrays, cv and rhoq. In figure 7, the arrows show the relationship between definitions and uses.

There are three steps to translating a CP from a use to a definition. First, where possible, we establish a one-to-one linear mapping from subscripts of the use to corresponding subscripts of the definition. For the case of the use cv(j-1), this corresponds to the mapping $[j]^{def} \rightarrow [j-1]^{use}$. (Note that the j on the left and j in the right refer to two different induction variables that just happen to have the same name.) If it is not possible to establish a 1-1 mapping for a particular subscript, or the mapping function is non-linear, this step is simply skipped. Next, we apply the inverse of this mapping to the subscripts in the on_home references in the CP of the use. This translates on_home lhs(i,j,k,2) to on_home lhs(i,j+1,k,2). Finally, any untranslated subscripts that remain in the CP from the use are vectorized through any loops surrounding the use that do not also enclose the definition. (There are no such subscripts in this example.)

Vectorization transforms a subscript that is a function of a loop induction variable into a range obtained by applying the subscript mapping to the loop range. The statement defining a privatizable array gets the union of the CPs translated from each use in this fashion. In figure 7, a translation is applied from the use CP to a definition CP along each of the red and green arrows. For a privatizable scalar definition, the process is simpler. There is no subscript translation to perform; CPs from each use are simply vectorized out through all loops not in common with the definition. The blue arrow represents a trivial vectorization (a simple copy) of the CP from the uses of the privatizable scalar ru1 to its definition since the use CPs are on statements in the same loop.

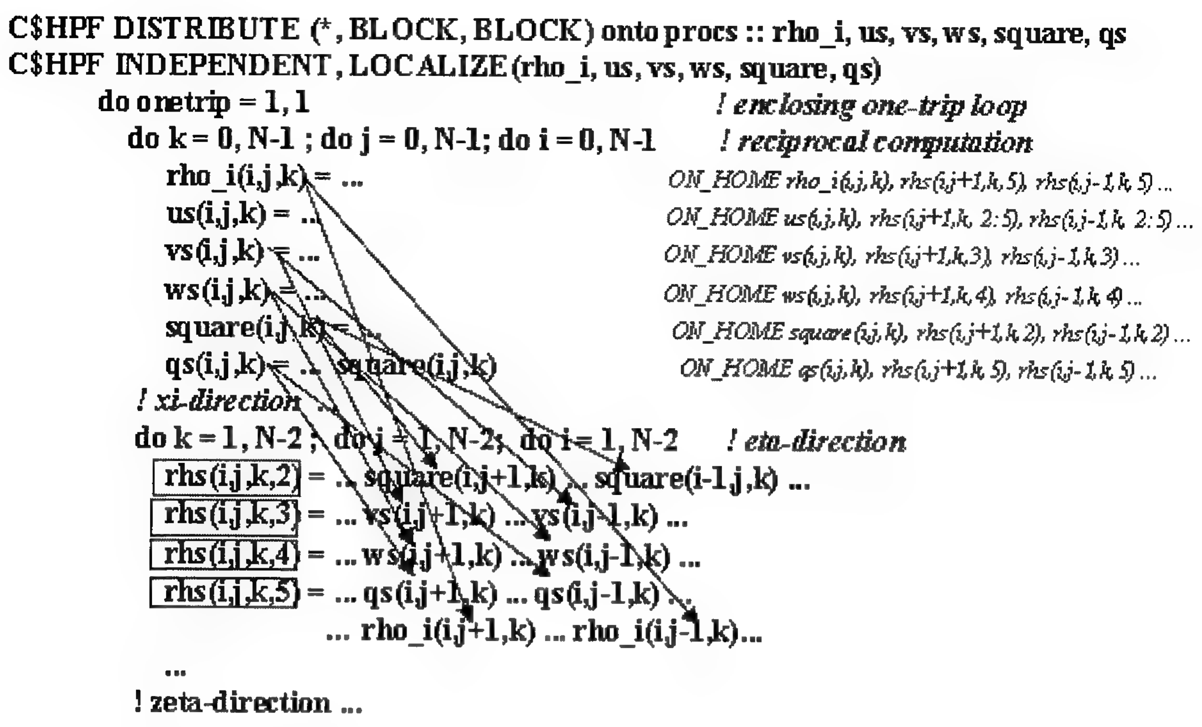
The effect of this CP propagation phase is that each boundary value of cv(j) and rhoq(j) is computed on both processors at either side of the boundary, therefore avoiding communication for these arrays within the i loop. All non-boundary values of cv and rhoq are computed only on the processor that owns them. Through this example, we have illustrated how our strategy avoids both costly communication inside the i loop, as well as needless replication of computation. It is worth noting that this CP propagation strategy for NEW variables is insensitive to the data layout of these variables. Regardless of what data layout directives for the NEW variables may be specified, CP propagation ensures that all and only the values that are needed on each processor are computed there.

3.4.3 Partial Replication of Computation

To support efficient stencil computations on distributed arrays, we developed support in dHPF for a new directive we call LOCALIZE [AJMCY98]. By marking a variable with LOCALIZE in an INDEPENDENT loop, the user asserts that all of the values of the distributed array that has been marked will be defined within the loop before they are used within the loop. Furthermore, the LOCALIZE directive signals the compiler that the assignment of any boundary values of the marked arrays that are needed by neighboring processors within the loop should be replicated onto those processors in addition to being computed by the owner. Unlike the NEW directive which requires that values assigned to a marked variable within the loop are not live outside the loop, variables marked with LOCALIZE for a loop may be live after the loop terminates. Marking a variable with LOCALIZE for a loop has the effect of ensuring that no communication of the marked variables will occur either during the loop or as part of the loop's finalization code. Instead, additional communication of values will occur before the loop to enable boundary value computations for the marked variables to be replicated as required to ensure that each use of a marked variable within the loop will have a local copy of the value available. To explain how the dHPF compiler uses its general CP model to implement such partial replication of computation, consider the loop nest shown in figure 8 from subroutine **compute_rhs** of the NAS 2.3-serial SP benchmark, with HPF directives added.

In **compute_rhs**, rhs is evaluated along each of the xi, eta, and zeta directions. Along each direction, flux differences are computed and then adjusted by adding fourth-order dissipation terms along that direction. To reduce number of operations (especially floating-point divisions) reciprocals are computed once, stored, and then used repeatedly as a multiplier at the expense of storage for the reciprocal variables rho_i, us, vs, ws, square , and qs. Without partial replication of computation, the best CP choice for each statement that defines a reciprocal would be **on_home** var(i,j,k) (where var stands for one of the reciprocal variables), and the statements that add dissipations would be **on_home** rhs(i,j,k,m) ($1 < m < 6$). However, this CP choice causes each subsequent reference to the reciprocal variables var(i,j-1,k) and var(i,j+1,k) in the rhs computation along the eta direction to become non-local. Similar references along the zeta direction become non-local as well. In this case, the boundary data of all these arrays would need to be communicated, which can be costly in distributed memory systems.

To reduce the cost of boundary communication for the reciprocal variables, we added an outer one trip loop to define a scope in which to LOCALIZE the reciprocal variables. In a fashion similar to how we calculate CPs for statements defining privatizable arrays, CPs for statements that define elements of arrays marked LOCALIZE are translated and propagated from statements that use these elements later. Consider the reciprocal variable rho_i in Figure 4.2. To calculate the CP for the statement defining elements of rho_i, we translate the **on_home** CP at each of the use sites and propagate it back to the definition site. For the xi direction flux computation we translate **on_home** rhs(i,j,k,5) to **on_home** rhs(i+1,j,k,5) from the use rho_i(i+1,j,k), and **on_home** rhs(i-1,j,k,5) from the use rho_i(i-1,j,k). The same translation process needs to be performed for uses in the eta and zeta directions. The CP of the statement that defines rho_i finally is set to the union of all the propagated CPs as well as the definition's owner-computes CP **on_home** rho_i(i,j,k).



```
C$HPF DISTRIBUTE (*, BLOCK, BLOCK) onto procs :: rho_i, us, vs, ws, square, qs
C$HPF INDEPENDENT, LOCALIZE(rho_i, us, vs, ws, square, qs)
        do onetrip = 1, 1                                  ! enclosing one-trip loop
          do k = 0, N-1 ; do j = 0, N-1; do i = 0, N-1     ! reciprocal computation
            rho_i(i,j,k) = ...                  ON_HOME rho_i(i,j,k), rhs(i,j+1,k,5), rhs(i,j-1,k,5) ...
            us(i,j,k) = ...                     ON_HOME us(i,j,k), rhs(i,j+1,k, 2:5), rhs(i,j-1,k, 2:5) ...
            vs(i,j,k) = ...                     ON_HOME vs(i,j,k), rhs(i,j+1,k,3), rhs(i,j-1,k,3) ...
            ws(i,j,k) = ...                     ON_HOME ws(i,j,k), rhs(i,j+1,k,4), rhs(i,j-1,k,4) ...
            square(i,j,k) = ...                 ON_HOME square(i,j,k), rhs(i,j+1,k,2), rhs(i,j-1,k,2) ...
            qs(i,j,k) = ... square(i,j,k)       ON_HOME qs(i,j,k), rhs(i,j+1,k,5), rhs(i,j-1,k,5) ...
          ! xi-direction ...
          do k = 1, N-2 ; do j = 1, N-2; do i = 1, N-2     ! eta-direction
            rhs(i,j,k,2) = ... square(i,j+1,k) ... square(i-1,j,k) ...
            rhs(i,j,k,3) = ... vs(i,j+1,k) ... vs(i,j-1,k) ...
            rhs(i,j,k,4) = ... ws(i,j+1,k) ... ws(i,j-1,k) ...
            rhs(i,j,k,5) = ... qs(i,j+1,k) ... qs(i,j-1,k) ...
                           ... rho_i(i,j+1,k) ... rho_i(i,j-1,k)...
            ...
          ! zeta-direction ...
```

Figure 8: Using LOCALIZE to partially replicate computation in subroutine compute_rhs from NAS SP.

Thus, the CP for the definition of rho_i becomes the union of on_home rhs(i+1,j,k,5), on_home rhs(i-1,j,k,5), on_home rhs(i,j+1,k,5), on_home rhs(i,j-1,k,5), on_home rhs(i,j,k+1,5), on_home rhs(i,j,k-1,5), and on_home rho_i(i,j,k). By partially replicating computation, computation of boundary data is executed not only on the processor which owns the data, but also on processors that need the data. As a result, the boundary communications of rho_i along the distributed directions can be eliminated. Similarly we can avoid the communication for us, vs, ws, square, and qs in the compute_rhs with the partial replication of computation. The dHPF compiler applies this same technique to subroutine compute_rhs in the NAS BT application benchmark.

Though partial replication of computation itself might introduce additional communication for the statements at the definition sites as an effect of the CP selection, it is beneficial when the number of messages and volume of the data to partially replicate the computation is smaller than transferring the values of the arrays marked LOCALIZE.

3.5 Code Generation for General Computation Partitions

A computation partitioning provides a specification of which dynamic instances of each statement must be executed by each processor. The dHPF compiler uses the computation partitioning specified for each statement as the basis for generating a statically-partitioned single program multiple data (SPMD) node program. Unlike previous partitioning models, the model used by dHPF supports partitioning of general control flow statements. As part of the dHPF project, we developed a code generation strategy that can support this more general partitioning model.

A key challenge to implementing computation partitionings specified by our model is to generate efficient code for loops. Since each statement in a loop can have a different partitioning, it may be necessary to split a loop into multiple convex sections to avoid expensive runtime checks. A key design choice is how to generate code for nested loops. A top-down strategy (in which outer loops are processed before the statements they enclose) can provide precise information about enclosing scopes because they will have already been partitioned. However, exploiting this precise information can be costly, requiring many more applications of the partitioning algorithm than a bottom-up strategy. For example, consider the cost of generating code for a triply nested loop for which each loop is eventually split into two sections. A top-down strategy would invoke the loop partitioning algorithm once for the outermost loop, twice for the middle loop, and four times for the inner loop. In contrast, a bottom-up strategy would invoke it only three times; however, at the inner levels, it would have no information about the two separate sections that will result at the outer levels.

We use a two-pass algorithm to resolve this tradeoff, using the more efficient bottom-up strategy to synthesize a correct (and fairly efficient) partitioned SPMD node program, and then use a simpler "top-down" algorithm to further simplify the resulting code. The second top-down pass uses contextual information to simplify control flow by using knowledge about enclosing scopes. We describe these two passes briefly below.

3.5.1 Realizing Computation Partitions

As the first step in code generation for realizing the computation partitionings specified by our model, we use a hierarchical, bottom-up code generation strategy [AMC98b]. Code generation proceeds in a depth first traversal over the tree of scopes in a procedure. For each scope, we compute a vector of sets, CPMap($\underline{m}$), one for each *statement group* in the scope. (A statement group is a sequence of consecutive statements with identical computation partitionings. $\{\underline{m}\}$ is a singleton set representing the processor index vector for the fixed processor *myid*.) We use Kelly, Pugh, and Rosser's algorithm for multiple-mappings code generation [KPR95] to compute loop bounds and guards that enumerate the SPMD iteration space for each statement group in the scope. To avoid adding the same guards at multiple levels, we treat the iteration set of the immediately enclosing scope statement as "known" information. This means that these constraints will be enforced when generating code for the enclosing scope and do not need to be enforced again for any scope nested inside.

To partition a procedure based on CP assignments, we perform a post-order traversal of a tree-based data abstraction of the procedure, applying code generation transformations as needed at each node in the tree. A node in the tree represents a single DO statement, a single branch of a conditional branch, or a sequence of simple statements. At each node in the tree, the code generation strategy to apply is selected independently. For loops, we currently support two strategies: simple bounds reduction, or loop-splitting (as described in Section 3.3.2) combined with bounds reduction for the individual loop sections. Other alternatives applicable to loops with irregular data layouts or references, namely runtime resolution and an inspector-executor can be added to this framework.

To partition computation for a sequence of statements with regular computation partitionings that can be represented explicitly as integer mappings, we use the Omega library's algorithm for code generation with multiple iteration spaces [KPR95]. This algorithm takes a vector of possibly non-convex) iteration spaces, each representing a statement, and uses it to synthesize a code template that enumerates the iteration space for each statement in lexicographic order. Figure 9 shows two iteration spaces and the corresponding code template produced by this code generation algorithm. In the code template, `s1` and `s2` represent placeholders for the statements represented by the first and second iteration space, respectively. While this code generation algoritjm is very powerful, it must be used carefully because it can be costly as its implementation is based on FME, which has doubly exponential complexity in the worst case [Sch86].

This code generation algorithm reduces loop bounds and can lift guards out of inner loops when

$\{[i,j] : 1 \le i \le 10 \ \wedge 1 \le j \le 5\}$

$\{[i,j] : 1 \le i \le 6 \ \wedge 1 \le j \le 5 \ \wedge cond \ge 4\}$

(a) Two iteration spaces.

```
for(t1 = 1; t1 <= 10; t1++)
  if (cond >= 4 && t1 <= 6)
    for(t2 = 1; t2 <= 5; t2++)
      s1(t1,t2);
      s2(t1,t2);
  if (t1 >= 7 && cond >= 4)
    for(t2 = 1; t2 <= 5; t2++)
      s1(t1,t2);
  if (cond <= 3)
    for(t2 = 1; t2 <= 5; t2++)
      s1(t1,t2);
```

(b) Code template generated using Omega's FME-based code generation.

Figure 9: Constructing a code template from iteration spaces.

```
do i = 1, N
  S1(i)
  do j = 1, M
    S2(i,j)
    S3(i,j)
  enddo
enddo
```

$$\begin{aligned}
LoopCP_1() &= \text{Project}(CP_1(i) \cup LoopCP_2(i),\ \{[i] : 1 \le i \le N\}) \\
CP_1(i) &= \{[i,j] : \text{myid owns } A_1(f_1(i))\} \\
LoopCP_2(i) &= \text{Project}(CP_2(i,j) \cup CP_3(i,j),\ \{[j] : 1 \le j \le M\}) \\
CP_2(i,j) &= \{[i,j] : \text{myid owns } A_2(f_2(i,j))\} \\
CP_3(i,j) &= \{[i,j] : \text{myid owns } A_3(f_3(i,j))\}
\end{aligned}$$

Figure 10: Example showing iteration sets constructed for code generation.

multiple statements with non-overlapping iteration spaces exist. An important step for simplifying the code we generate for a scope is to provide available information about enclosing scopes. For example, consider the loop nest in Figure 10. To generating code for the inner (j) loop alone, we fix the value of i at a symbolic value I. We then use $LoopCP_2(I)$ as "known" information because the constraints in $LoopCP_2(i)$ will be enforced when partitioning the enclosing i loop. Therefore, the inner loop is partitioned for the vector of CPs $[CP_2(I,j), CP_3(I,j)]$ with $LoopCP_2(I))$ as known information. Then, the outer loop is partitioned for the vector of CPs $[CP_1(j), LoopCP_2(i)]$ with $LoopCP_1()$ as known information.

We use the example source loop nest in Figure 11 to illustrate the dHPF compiler's bottom-up computation partitioning code generation strategy for compiling an HPF program to a message-passing system. This code is a fragment abstracted from a pipelined computation in the Erlebacher benchmark, a derivative calculation using an implicit sixth-order compact-differencing scheme.[4] Comments in the figure show the computation partitioning (CP) and the initial placement of communication operations chosen automatically by the dHPF compiler. Figure 12 shows the corresponding intermediate code generated by dHPF. (For brevity, Figure 12 uses "COMPUTE `f(1:64, j, k)`" to represent a copy of the `i` loop.) The innermost statement is assigned to be executed by the processor that owns `f(i,j,k)`, as defined by the alignment and distribution of the array `f`. This results in the `k` loop being partitioned among the processors. The reference `f(i,j,k+1)` therefore accesses non-local data, and the required communication is placed inside the `k` loop since the communicated data is modified inside this loop.

During code generation, communication is represented simply by placeholders for SEND and

[4]This benchmark was developed by Thomas Eidson at ICASE.

```
      parameter (N=64)
      real c(N), f(N,N,N)
CHPF$ processors p(4)
CHPF$ distribute f(*,*,block) onto p
CHPF$ distribute     c(block) onto p
      do j=1,N
        do k=N-1,1,-1
C         SEND f(1:N,j,k+1)                       ! ON_HOME f(1:N,j,k+1)
C         RECV f(1:N,j,k+1)                       ! ON_HOME f(1:N,j,k)
          do i=1,N
            f(i,j,k) = f(i,j,k) - c(k) * f(i,j,k+1)   ! ON_HOME f(i,j,k)
```

Figure 11: HPF source fragment abstracted from the Erlebacher benchmark.

```
     do j = 1, 64
1      if (pmyid1 <= 2) then
2        k = 16 * pmyid1 + 16
         !--<< Iterations that access only local values >>--
3        if (16 * pmyid1 >= k - 15) then                  !UNSATISFIABLE
           COMPUTE f(1:64, j, k)
4        if (16 * pmyid1 == k - 16  && pmyid1 <= 2) then  !TAUTOLOGY
           RECV f(1:64, j, 16 * pmyid1 + 17)
         !--<< Iterations that read non-local values >>--
5        if (16 * pmyid1 <= k - 16) then                  !TAUTOLOGY
           COMPUTE f(1:64, j, k)
6      do k = 16 * pmyid1 + 15, 16 * pmyid1 + 1, -1
7        if (16 * pmyid1 == k &&  pmyid1 >= 1) then       !UNSATISFIABLE
           SEND f(1:64, j, 16 * p_myid1 + 1)
         !--<< Iterations that access only local values >>--
8        if (16 * pmyid1 >= k - 15) then                  !TAUTOLOGY
           COMPUTE f(1:64, j, k)
9        if (16 * pmyid1 == k - 16 && pmyid1 <= 2) then   !UNSATISFIABLE
           RECV f(1:64, j, 16 * pmyid1 + 17)
         !--<< Iterations that read non-local values >>--
10       if (16 * pmyid1 <= k - 16) then                  !UNSATISFIABLE
           COMPUTE f(1:64, j, k)
       if (pmyid1 >= 1) then
         k = 16 * pmyid1
11       if (16 * pmyid1 == k && pmyid1 >= 1) then        !TAUTOLOGY
           SEND f(1:64, j, k + 1)
```

Figure 12: Skeletal SPMD code for Fig. 11 with partitioned computation.

RECV statements. The compiler assigns appropriate CPs to these placeholders to ensure the communication is executed by the required processors. These CPs simply specify that the SEND is to be executed by the owner of `f(i,j,k+1)` and the RECV by the reader of the data, viz. the owner of `f(i,j,k)` (for $1 \leq i \leq N$). Note that these communication CPs are conservative because the

SEND and RECV should actually execute only in a subset of the iterations of the k loop, namely in the "boundary" iterations on each processor.[5] We rely on conditionals in the communication code instantiated for the SEND and RECV placeholders to enforce precisely when communication needs to occur.

Here we briefly describe two key steps in performing the computation partition code generation for this example:

1. *Non-local index-set splitting on the* `i` *loop*: This step splits the innermost loop to separate the iterations that access only local data from those that access non-local data. The motivation for this splitting operation is twofold. First, it enables data communication for non-local iterations to be overlapped with the computation of local iterations. Second, it is desirable to access non-local data directly out of communication buffers (to avoid data copies) while still accessing local data in-place; splitting minimizes the execution frequency of ownership guards needed to select between the local and non-local storage areas. (These guards are not shown in this example.)

   Splitting the `i` loop results in two full copies of the loop with different conditions on `k` ($\texttt{k} \leq 16 * \texttt{pmyid1} + 15$ and $\texttt{k} \geq 16 * \texttt{pmyid1} + 16$ on lines 8 and 10 respectively). During this step, the only information available about the enclosing context is that the bounds of the `k` loop will be reduced according to the CPs of its enclosed statements, i.e., $16 * \texttt{pmyid1} \leq \texttt{k} \leq 16 * \texttt{pmyid1} + 16$.

2. *Loop bounds reduction with guard lifting on the* `k` *loop*: This step reduces the loop bounds of the `k` loop to partition the iterations among processors as specified by the CPs assigned to the SEND, COMPUTE, and RECV statements. Since the SEND should not execute in the first iteration of each processor and the RECV and COMPUTE blocks should not execute in the last, the loop is fragmented into three sections to avoid introducing additional ownership guards within the `k` loop. The resulting loop sections are $\{\texttt{k} = 16 * \texttt{pmyid1} + 16\}$, $\{16 * \texttt{pmyid1} + 15 \geq \texttt{k} \geq 16 * \texttt{pmyid1} + 1\}$, and $\{\texttt{k} = 16 * \texttt{pmyid1}\}$. The COMPUTE blocks for the two sections of the `i` loop (local and non-local) are simply replicated in the first two sections of the `k` loop (lines 3,5 and 8,10). Similarly, the RECV and SEND are replicated as shown. The aforementioned fragmentation of the `k` loop results in refined contexts for copies of the `i` loop code inserted into each fragment. As a result of this fragmentation, the conditions on lines 3, 7, 9 and 10 are unsatisfiable, and those on lines 4, 5, 8 and 11 are guaranteed true as noted in figure 12.

The index-set splitting [AMC98b, KM91] in step 1 and guard lifting in step 2 described above are sophisticated transformations that aim to minimize the dynamic frequency of conditionals and otherwise improving the efficiency of generated code. These transformations have the effect of splitting a loop into more refined contexts. Since we generate code for loops bottom up, these refined contexts are not exploited in this phase. However, the refined contexts created introduce many opportunities for simplifying guards generated for inner fragments. Later code generation steps that produce code for communication operations and add ownership guards for references also are performed without contextual knowledge. In the following section, we describe how we exploit contextual knowledge in a top-down pass to eliminate unnecessary guards rendered unnecessary.

[5]Communication CPs are conservative only when communication happens inside a partitioned loop. In such cases, precise CPs for communication are difficult to compute and express in any general manner since communication patterns can be quite complex.

3.5.2 Constraint Propagation and Control Flow Simplification

Here we describe a new global optimization strategy developed for the dHPF compiler to eliminate redundant conditional control flow generated in the course of compiling data-parallel programs [MCA97, MCA98]. As described in the previous section, applying our our bottom-up code generation algorithm to loops refines the context they present to code fragments nested inside. This refinement of enclosing context for a code fragment can cause some conditionals in the fragment to become wholly or partly tautological or unsatisfiable, rendering them unnecessary. Applying our top-down code optimization strategy to SPMD code generated by the dHPF compiler for three benchmarks of varying complexity showed that our approach is consistently highly effective at simplifying the conditional control flow once contextual knowledge is available. For the benchmarks we studied, our control-flow simplification strategy eliminated 35% to 67% of the `if` statements that were present after SPMD code generation.

Our control-flow simplification optimization is performed in three phases. The first phase analyzes the initial SPMD code generated by dHPF during a prior (bottom-up) code generation pass. This analysis phase collects symbolic constraints on the values of integer expressions imposed by loops, conditional branches, assertions, and integer computations, and propagates these constraints over the control dependence graph. The constraints we compute for each point p in the program are conservative approximations of its *path condition*, which specifies the precise conditions under which p will be reached during execution. The second phase uses propagates these constraints, as appropriate, throughout the program. The third phase uses the propagated constraints to eliminate or simplify conditional tests whose outcome can be determined (in full or in part) at compile time, and simplifies the control flow based on those tests.

While we use this strategy for eliminating superfluous conditional control flow in SPMD code, it could also be used for other applications such as debugging [Bou93], optimizing embedded system software [Joh86], optimizing array bounds checking [Har77, KW95], improving dependence analysis [BE95], array privatization [BE95, TP95], and constant propagation [BTC95].

Our constraint propagation and simplification strategy leverages three key program analysis technologies: control dependence analysis [CFR+91], global value numbering based on static single assignment form [CFR+91, Hav94], and simplification of symbolic integer constraints expressed as formulae in Presburger arithmetic [KMP+96, Pug92]. (A Presburger formula consists of arbitrary affine inequalities and equalities of integer variables, together with the logical operations $\neg$, $\vee$, $\wedge$, and the quantifiers $\exists$ and $\forall$. For example, given a loop from 1 to N with a stride of 2, the constraints on the loop index variable i within the loop can be expressed as $\{\exists s : i = 2s + 1 \wedge 1 \leq i \leq N\}$.)

Below we describe each of the three phases of our control-flow simplification strategy.

Computing Constraints. We compute constraints that arise from conditional branches, loop bounds, and explicit constraint assertions which specify arbitrary logical expressions in terms of program variables. At the lowest level, we represent these constraints directly as Presburger formulae, as described earlier. Each variable in an inequality or equality (other than those that are existentially quantified) represents a value number of one of the following types: (a) an atomic program variable, (b) non-affine expressions of program variables, or (c) merged values such as the value for a variable on exit from a loop or conditional branch.

As an example, consider how we compute constraints enforced by the `if` statement in the source fragment below.

```
i = 2 * m
if (i * j + 8 .ge. i * 7 - 4 .and. intmod(i, 4) .eq. 1)
```

In this example, $\texttt{intmod(i,n)} \equiv i - n(i \text{ div } n)$, so that $0 \leq \texttt{intmod(i,n)} \leq n-1$. First, we compute a value number for the logical expression in the conditional test. In computing the value number for this expression, i gets replaced with `2m` everywhere. The first term of the resulting value number expression then is $2 * V_m * V_j + 8 \ .ge.\ 14 * V_m - 4$, where V_m and V_j represent the value numbers for the variables `m` and `j`. In translating this value number into a Presburger formula, since Presburger arithmetic only accommodates affine relations, we collapse the symbolic product $V_m * V_j$ into a single Presburger variable V_{mj}. The resulting constraint thus becomes $\{2V_{mj} + 12 \geq 14V_m\}$.

In the second term, the expression `intmod(i,4) .eq. 1` is translated as $\{\exists e : 0 \leq i - 4e \leq 3 \wedge (i - 4e) = 1\}$. After substituting i with $2V_m$ in this term and combining with the first term's constraints, the constraints for the full logical expression become:

$$\{2V_{mj} + 12 \geq 14V_m \wedge (\exists e : 0 \leq 2V_m - 4e \leq 3 \wedge 2V_m - 4e = 1)\}$$

Constraints for a loop are similarly computed by constructing a formula that constrains the index variable range according to the loop bounds and stride. One restriction is that a symbolic (i.e., unknown) stride cannot be represented precisely using Presburger formulas because it would require a product of symbolic terms. Similarly, an `intmod` or `div` with a symbolic divisor cannot be expanded as above, and thus must be treated as a non-affine expression, i.e., represented with a single Presburger variable. For such cases, we could assume compute a formula representing only the bounds constraints, which is a conservative assumption for our constraint analysis.

After constraint simplification, we convert a satisfiable constraint formula for a conditional back into an equivalent Fortran logical expression with Kelly, Pugh, and Rosser's code generation algorithm [KPR95]. After code generation, non-affine expressions are then substituted back into the generated code (e.g., V_{mj} is replaced by `m*j`).

Constraint Propagation. We developed a one-pass algorithm for propagating constraints on value numbers for program variables throughout a program. A one-pass algorithm because our value-number-based constraint representation allows us to ignore any constraints that would reach a statement along a back edge in the control flow graph.[6] In particular, a variable that is modified inside a loop is assigned a new value number at the top of the loop that represents the merge of the value number for the variable entering the loop from the outside, and the value number from the previous iteration. Constraints on the value number entering the loop from the outside need not be invalidated inside the loop because all uses of the loop-variant variable in the loop will refer to one or more different value numbers defined in the body of the loop.

By ignoring constraints along back edges, however, our propagation algorithm cannot directly compute constraints for iterative values defined in loops. Instead, for simple iterative values (e.g. an auxiliary induction variable that is a linear function of a loop index variable) our value numbering package computes symbolic range information which provides us with the appropriate constraints for these iterative values. We have not found it necessary to compute constraints on more complex iterative constructs.

Because we can safely ignore constraints along backward control flow edges, we perform our constraint propagation on the control-dependence graph (CDG), which is a natural representation for logical constraint propagation. A node n in a control flow graph is control dependent on another node b if and only if there is a path from b to the exit node that does not execute n, and an outgoing

[6]The algorithms we present assume that the control flow graph is reducible [Tar74], i.e., for a node within a natural loop, every path from the root to the node must include the loop header. Handling irreducible graphs correctly is straightforward: it is only necessary to avoid propagating constraints into or out of any nodes within an irreducible subgraph.

edge from b that (if taken) is guaranteed to execute n. The node b must be a branch node, and the CDG will include an edge from b to n. A statement inside an ordinary loop (without loop exits) has a single control dependence predecessor, i.e., the CDG has a single edge from the loop header node to the statement. Therefore, ordinary loops do not cause cycles in the CDG, and so backward control flow in ordinary loops is automatically ignored with this representation. Non-trivial back edges (forming a cycle) in the CDG occur due to constructs such as a jump out of a loop, but these too can be safely ignored as described above because any references to loop variant values inside the loop will refer to different value numbers than those entering the loop with constraints along forward control dependence edges.

The inputs to our algorithm for computing path constraints at each node in the program are the constraints imposed by conditional branches, loop bounds and strides, and assertions. Separate outgoing constraints are computed for each type of outgoing edge at a conditional branch (*i.e.*, TRUE and FALSE edges for `if` statements and ENTER and FALLTHROUGH edged for loops). We begin by initializing the incoming and outgoing constraints at each branch to `false`, and the assertions that apply to each outgoing CD edge type to `true`.

Next, we annotate CD edges with constraints from logical assertions. An assertion's constraints apply to any statement that is predominated by the assertion as long as it is prior to any redefinition of any of the variables in the assertion. However, once we translate an assertion expression into constraints in terms of variable value numbers, the constraints are globally true throughout the entire program and can be safely applied at any statement. We exploit this property as follows to treat assertions uniformly with conditional branch expressions. We apply the constraints from an assertion s to all statements that have the same control dependence relationship with a common CD parent, even though this includes statements that precede s. In other words, for each statement b with CD edge $b \rightarrow s$ of type t, we associate the assertion constraints with all CD edges of type t emanating from b. A second phase propagates these assertion constraints forward along the CD edges to which they apply.

The final phase of the propagation computes incoming and outgoing path constraints at each conditional branch node. These constraints are computed for each node in a reverse post order traversal. In reverse-post-order, all control dependence predecessors of a node along forward edges are visited before the node itself. This evaluation order enables transitive constraints to be propagated along all forward control dependence edges. At each conditional branch node b, we compute the incoming path constraints that hold when b is reached along any path. We compute these as a disjunction of the constraints along each incoming control dependence edge. The incoming constraints along a control dependence edge are simply the intersection of both the outgoing and assertion constraints from b's predecessor for that type of edge. Predecessors across back edges will contribute outgoing constraints of `false` (the identity element for logical disjunctions) since they have not been visited yet. As described earlier in this section, it is safe for us to ignore constraints along CD back edges; our initialization accomplishes this simply. Next, for each CD edge type leaving b, we set the default outgoing path constraints to the incoming path constraints for b. Finally, depending on b's node type, we fold the local constraints enforced by b into the different types of outgoing control dependence edge types as appropriate.

Control-flow Simplification. We use constraints computed at each conditional branch node to simplify a procedure's control flow. If the outgoing edge constraints for a loop entry or the `true` branch of a conditional are unsatisfiable, we eliminate its code. In the case of an IF, the entire IF statement is replaced with the statements in the `false` branch if any. For conditionals, one of two further simplifications is possible. If the incoming constraints at a logical IF are as

```
do j = 1, 64
  if (pmyid1 <= 2) then
    k = 16 * pmyid1 + 16
    RECV f(1:64, j, k + 1)
    !--<< Iterations that read non-local values >>--
    COMPUTE f(1:64, j, k)
  do k = 16 * pmyid1 + 15, 16 * pmyid1 + 1, - 1
    !--<< Iterations that access only local values >>--
    COMPUTE f(1:64, j, k)
  if (pmyid1 >= 1) then
    k = 16 * pmyid1
    SEND f(1:64, j, k + 1)
```

Figure 13: Skeletal SPMD code for Fig. 12 after simplification.

strict as the outgoing constraints on the **true** branch, the conditional will always evaluate to **true** when reached. Therefore, we replace the entire IF statement by statements in the **true** branch. Otherwise, we can try to simplify the guard condition by eliminating those constraints in the guard that are guaranteed by the incoming constraints. For two constraint formulae, f_1 and f_2, the operation $\texttt{Gist}(f_1, f_2)$ computes a (possibly simpler) set of inequalities f such that $f \wedge f_2 \Rightarrow f_1$. Applying the Gist operation to the outgoing constraints given the incoming constraints returns a conservative approximation of the non-redundant constraints. We then regenerate a new guard using the simplified constraints.

Effectiveness of Control Flow Simplification. After applying our simplification algorithm, the code shown in Figure 12 is reduced to the code shown in Figure 13. All of the infeasible branches (due to unsatisfiable or tautological guards) have been discovered and eliminated. In fact, numerous other guards (not shown in the figure) that were introduced as part of generating the communication for the SEND and RECV placeholders shown in figure 12 are also significantly simplified or eliminated by this process.

Experiments we performed (described in detail elsewhere [MCA98] indicate that control-flow simplification based on constraint propagation consistently provides substantial improvements in the number and complexity of conditional branches. Applied to SPMD code generated by dHPF for a set of benchmarks (Tomcatv, Erlebacher, Jacobi), it eliminated 35%–67% of the if statements in the generated code, reducing overall code size by 19%–34%. While most of these guards are not in loops in dHPF-generated code because of our use of aggressive loop optimization techniques such as splitting and guard lifting, our experiments showed that it added a few percent to performance, even for these simple programs. For programs with more complex communication, we expect the impact to be greater.

It is interesting to note that the final placement of the communication code in Figure 13 achieves the effect of an optimization known as vector message-pipelining [Tse93]. This optimization moves pipelined shift communication occuring in boundary iterations of a partitioned loop out of the loop to eliminate conditionals around the communication statements. This optimization is complex to implement in a general way using pattern-based techniques because simply lifting communication out of the loop can lead to deadlock when the loop body requires both forward and reverse pipeline communication. We safely achieve the effect of vector message pipelining through the combined effects of our CP code generation followed by control-flow simplification. Our code generation is safe because we place communication in the proper iterations to satisfy the pipeline dependences and our code generation strategy preserves lexical statement order as it transforms iteration spaces.

3.6 Experimental Evaluation of dHPF Compilation Techniques

A pair of computational fluid dynamics benchmarks from NASA served as driving applications for the dHPF compiler research. These codes helped focus the project's research on compiler analysis and code generation techniques to address the challenges of semi-automatic parallelization of realistic scientific codes using High Performance Fortran compilers.

As described in a NASA Ames technical report [BHS$^+$95], the NAS benchmarks BT and SP are two simulated CFD applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [BHS$^+$95]. Both consist of an initialization phase followed by iterative computations over time steps. In each time step, boundary conditions are first calculated. Then the right hand sides of the equations are calculated. Next, banded systems are solved in three computationally-intensive bi-directional sweeps along $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$ directions. Finally, flow variables are updated. Values must be communicated among processors during the setup phase and the forward and backward sweeps along each dimension.

If the arrays are distributed in block fashion in one or more dimensions, the bi-directional sweeps along each distributed dimension introduce both serialization and intensive fine-grain communication. To retain a high degree of parallelism and enable coarse-grain communication along all dimensions, the NPB2.3b2 versions of BT and SP solve these systems using a skewed block distribution called multi-partitioning [BHS$^+$95, Nai92, NNN93]. With multi-partitioning, each processor handles several disjoint subblocks in the data domain. The subblocks are assigned to the processors so that there is an even distribution of work among the processors for each directional sweep, and that each processor has a subblock on which it can compute in each step of the sweep. Thus, the multi-partitioning scheme maintains good load balance and coarse grained communication.

The serial version of the NPB2.3 suite[7], NPB2.3-serial, is intended to be a starting point for the development of both shared memory and distributed memory versions of these codes for a variety of hardware platforms, for testing parallelization tools, and also as single processor benchmarks. The serial versions were derived from the MPI parallel versions of the benchmarks by eliminating the communication and having one processor compute the entire data domain. The serial versions of the code are nearly identical to their parallel counterparts, except for the lack of data domain partitioning and explicit communication. Therefore, the MPI version provides an excellent basis for evaluating the performance of compiler parallelization of the serial benchmarks. Both the hand-written and compiler-generated codes use non-blocking send/recv for communication.

We studied the serial versions of the NAS application benchmarks to understand the challenges that arise in real-world applications and develop compiler techniques to address them. Our work with these benchmarks was a key motivator behind our work on computation partition selection for loop nests that use privatizable arrays, commmunication-sensitive loop distribution (to eliminate inner-loop communication without excessively sacrificing cache locality), interprocedural selection of computation partitions, and data availability analysis to avoid excess communication for values computed but not owned by the local processor.

[7]The NAS benchmarks are available from `http://science.nas.nasa.gov/Software/NPB`.

We evaluated performance of both the dHPF compiler itself and the code it generates. We compared the code generated by the dHPF compiler with hand-written MPI and with code generated by the PGI pghpf compiler [AJMCY98]. Starting with minimally modified versions of the NAS2.3 serial codes plus HPF directives, our techniques are able to achieve performance within 15% of the hand-written MPI code for BT and within 21% for SP on 25 processors. The code generated by the dHPF compiler also outperforms the code generated by the PGHPF compiler for most cases, even though the HPF code used with dHPF was largely identical to the original serial version of the benchmarks. Compile-times for large application code are somewhat high but we believe it is acceptable and can be substantially improved.

Section 3.7 describes ongoing work to obtain significantly better scalability and performance by having the dHPF compiler generate code using a multipartitioning strategy for partitioning data and computation. The experiments described below study codes using standard HPF block partitionings.

3.6.1 Compiler Performance

We assessed the performance of the dHPF compiler in terms of the detailed costs of applying the set-based compilation techniques described in previous sections. The benchmarks we use are TOMCATV—a SPEC92 benchmark that performs mesh generation, and a serial verion of NAS SP. Both programs perform stencil-based computations on multidimensional arrays.

The version of TOMCATV we studied is simply the Fortran 77 code for the SPEC92 benchmark with HPF directives to specify a `(BLOCK, *)` distribution of the arrays over a 1D processor grid. With our directives added, this benchmark has 228 lines and a single procedure. Compared to TOMCATV, the SP application benchmark is more than an order of magnitude larger and the computation is considerably more complex. SP has much larger and non-uniform loop nests, procedure calls within parallel loops, and makes liberal use of privatizable arrays whereas TOMCATV uses only privatizable scalars. We developed an HPF version of SP using minimal modifications to the serial Fortran 77 code from the NPB2.3-serial release. We specified *block* distributions in the `y` and `z` spatial dimensions of the program's 3D and 4D arrays. Our modified version of the source is 3502 lines, compared to the 3382 lines in the NPB2.3-serial release. The application has 30 procedures.

We used a version of dHPF compiled with `-O2` optimization and measured compile times on a 250MHz UltraSparc workstation. We used Rational Software's Quantify^{TM} utility to obtain an execution profile. For TOMCATV, the number of processors was left unspecified at compile-time. For SP, we considered two variants: SP-4 used a fixed 2x2 processor array, while SP-sym left the total number unspecified, using a `2 x (number_of_processors()/2)` processor array. For both benchmarks, the compiler exploited all of the optimizations described in previous sections, including non-owner-computes computation partitionings for some statements to reduce the number and frequency of communication operations.

Table 1 shows the wall-clock time in seconds for compiling each of these benchmark versions and a breakdown of total execution time spent in key phases of dHPF, including each of the major integer-set optimizations. (The final optimization of generated code does not use integer-set-based operations.) The nesting of phases is shown by indentation in the first column. The numbers do not sum to 100% because percentages shown for indented phases are merely refinements of their enclosing phase. Although the benchmarks are quite different in size and complexity, the breakdown of compilation time for them is remarkably consistent. None of the phases is especially dominant in compile time, although SP has a somewhat high cost for communication generation because of a huge number number of communications (even after a high degree of coalescing), some with

Breakdown of compilation time			
application	SP-4	SP-sym	T-sym
total compilation wall-clock time	1145s	1073s	28s
interprocedural analysis	1.2%	1.4%	1.5%
module compilation	97.9%	97.8%	97.1%
partitioning computation	14.5%	11.3%	16.1%
loop splitting	6.4%	2.0%	3.7%
loop bounds reduction	5.6%	6.7%	6.1%
communication generation	31.4%	34.6%	28.1%
loops to compute msg sizes	12.9%	13.4%	7.0%
loops over comm partners	12.8%	14.1%	10.4%
check if msg is contiguous	1.3%	1.9%	2.6%
check if msg is rect section	1.2%	1.4%	1.4%
opt of generated code	28.1%	28.9%	21.3%
mult mappings code generation	26.4%	23.9%	10.5%

Table 1: Breakdown of dHPF compilation time.

complex patterns. Based on these results, we would anticipate that other programs would have a similar cost breakdown although the distribution of compilation effort would differ depending on the number and complexity of loops in a routine, the number of communication events, and the shape of the communication sets.

At the bottom of the table, we note the time spent in Kelly, Pugh and Rosser's multiple mappings code generation operation which we use to synthesize loops that enumerate iteration or data sets. This algorithm accounts for virtually all of the cost of the set framework. Most of this time is spent in simplifying Presburger formulae representing integer sets and mappings. These numbers show that the integer-set framework is not a dominant cost in compile-time, even for fairly large and complex codes such as SP. Finally, the table shows that there is no significant additional cost to compiling for a symbolic number of processors vs. a known (fixed) number. SP-sym is in fact faster than SP-4 because the compiler performs more aggressive loop-splitting in the latter, which leads to more complex sets and correspondingly higher compile-time cost.

The absolute compile-times for SP are somewhat high but (we believe) acceptable for a research compiler where efficiency in the implementation has not been a primary goal. There are opportunities for significant improvements. Approximately 30% of compilation time is spent on generating custom inline code for counting, packing and unpacking buffers. While such custom code can be important for complex communication patterns, for the vast majority of simple patterns such as a shift communications, invoking a run-time library operation would not be more expensive, and would largely eliminate the communication generation cost. We also spend nearly 30% in a post-pass optimizing the SPMD code we generate, which we believe can be largely eliminated through an algorithmic improvement.

3.6.2 Performance of dHPF-generated code

We compared the performance of compiler-parallelized versions of the NAS benchmarks SP and BT against the hand-written MPI versions of the benchmarks from the NPB2.3b2 release. The compiler-generated parallel codes for these benchmarks were produced by dHPF and pghpf (a commercial product from the Portland Group). We used different HPF versions of the codes for the pghpf and the dHPF compilers. In particular, each compiler was applied to HPF codes created by the compiler's developers to best exploit its compiler's capabilities. For the dHPF compiler, we created HPF versions of the SP and BT benchmarks by making small modifications to the serial Fortran 77 versions of the benchmarks in the NPB2.3-serial release. We modified the code primarily to add HPF directives and to interchange a few loops to increase the granularity of the parallelization. (More detailed descriptions of these modifications appear below.) With the pghpf compiler, we used HPF versions of the SP and BT benchmarks that were developed by PGI. The PGI HPF implementations are derived from an earlier version of the NAS parallel benchmarks written in Fortran 77 using explicit message-passing communication. PGI rewrote these into a Fortran 90 style HPF implementation in which communication is implicit. (The PGI HPF versions were developed before the NPB2.3-serial release was available.)

The experimental platform for our measurements was an IBM SP2 running AIX 4.1.5. All experiments were submitted using IBM's poe under control of LoadLeveler 3.0 on a block of 32 120Mz P2SC "thin" nodes that was managed as a single-user queue. Loadleveler was configured to only run one process per processor at any time. PGI-written HPF codes were compiled using pghpf version 2.2. All codes - the hand-written NPB MPI codes and the codes generated by pghpf and dHPF - were compiled with IBM's xlf Fortran compiler with command line options `-O3 -qarch=pwr2 -qtune=pwr2 -bmaxdata:0x60000000`. All codes were linked against the IBM MPI user space communication library with communication subsystem interrupts enabled during execution.

Our principal performance comparisons are on 4, 9, 16, and 25 processor configurations because the hand-written codes in the NAS 2.3b2 release require a square number of processors. We also present the performance for PGI and dHPF generated code on 2, 8, 27 and 32 processors for reference. Single-processor measurements (and in some cases 2- and 4-processor measurements) were not possible because the per-node memory requirements exceeded the available node memory on our system.

In the following subsections, we compare the performance of the hand-written MPI against the compiler-generated code for SP and BT. For each of the benchmarks, we present data for Class A and Class B problem sizes (as defined by the NAS 2.0 benchmarking standards). We present the raw execution times for each version and data size of the benchmarks along with two derived metrics: relative speedup[8], and relative efficiency[9]. Since we were unable to measure single processor execution times, we lack the basis for true speedup measurements. Instead, based on our experience with the sequential and hand-coded MPI versions on SP2 "wide" nodes that contain more memory, we assume that the speedup of the 4-processor version of the hand-coded MPI programs is perfect (which is approximately true). All speedup numbers we present are computed relative to the time of the 4-processor hand-coded versions. Our relative efficiency metric compares the relative speedup of the dHPF and pghpf generated codes against the speedups of the corresponding hand-written versions. This metric directly measures how much of the performance of the hand-coded benchmarks is achieved by the compiler-parallelized HPF codes. As described above, the hand-written MPI parallelizations of both the SP and BT benchmarks use a skewed block data distribution known as multi-partitioning. This partitioning ensures that each processor has a subblock to compute at each step of a bi-directional line-sweep along any spatial dimensions. This leads to high efficiency because each processor begins working on a subblock immediately as a sweep begins without needing to wait for partial results from another processor. The multi-partitioning distribution is not expressible in HPF, and therefore the HPF implementations incur added costs in communication, loss of parallelism, or both.

[8]Speedups are relative to the 4-processor hand-written code for Class A and 16-processor hand-written code for Class B, which are assumed here to have perfect speedup.

[9]Relative efficiency is computed by comparing speedup of dHPF generated code with its hand-written counterpart.

Instead of multi-partitioning, the PGI and Rice HPF implementations of SP and BT use block distributions. We describe the key features of the two HPF implementations here to provide a basis for understanding the compiler-based parallelization approach used by the dHPF and pghpf compiler versions and its overall performance.

3.6.3 SP

In developing the Rice HPF implementation of SP from the NPB2.3-serial release, our changes to the serial code amounted to 147 of 3152 lines or 4.7%. The breakdown of our key changes is as follows:

- Removed array dimension (cache) padding for arrays **u**, **us**, **vs**, **ws**, **forcing**, **qs**, **rho_i**, **rhs**, **lhs**, **square**, **ainv**, and **speed**. The cache padding interfered with even distribution of work in the HPF program. Instead, the dHPF compiler automatically pads arrays in the generated code to make all array dimension sizes odd numbers.
- Eliminated the **work_1d** common block. Variables **cv**, **rhon**, **rhos**, **rhoq**, **cuf**, **q**, **ue**, **buf** were instead declared as local variables where needed in the **lhsx**, **lhsy**, **lhsz** and **exact_rhs** subroutines.
- Added HPF data layout directives to specify a `BLOCK,BLOCK` distribution of the common arrays (**u**, **us**, **vs**, **ws**, **forcing**, **qs**, **rho_i**, **rhs**, **lhs**, **square**, **ainv**, **speed**) in the **y** and **z** spatial dimensions.
- Added 6 HPF `INDEPENDENT NEW` directives. In the **lhsx**, **lhsy**, and **lhsz** subroutines, a `NEW` directive was used to identify two array as privatizable in the first loop nest. In the **exact_rhs** subroutine, three `NEW` directives were used to specify **cuf**, **buf**, **ue**, **q**, and **dtemp** as privatizable in each of three loop nests.
- In the **compute_rhs** subroutine, we added an outer one-trip loop along with an `INDEPENDENT LOCALIZE` directive (a dHPF extension to HPF) for the **rho_i**, **square**, **qs**, **us**, **ws**, and **vs** arrays. This directive has the effect of eliminating communication inside a loop for the specified variables by partially replicating computation of these variables inside the loop so that each element of these arrays will be computed on each processor on which it is used.
- Inlined 2 calls to **exact_solution** in subroutine **exact_rhs** where our interprocedural computation partitioning analysis was (currently) incapable of identifying that a computation producing a result in a privatizable array should be treated completely parallel.
- Interchanged loops to increase the granularity of computation inside loops with carried data dependences to increase the granularity of computation and communication for two loop nests in subroutine **y_solve** and 4 loop nests in subroutine **z_solve**.

The HPF `INDEPENDENT` directives used were not for the purpose of identifying parallel loops because dHPF automatically detects parallelism in the original sequential Fortran 77 loops. The only reason HPF `INDEPENDENT` directives were added to some loops in the code was to specify privatizable variables with the `NEW` directive and variables suitable for partial replication of computation with the `LOCALIZE` directive. With the data partitionings in the **y** and **z** dimensions, only partial parallelism is possible in the **y** and **z** line solves because of the processor-crossing data dependences. The dHPF compiler exploits wavefront parallelism in these solves using non-owner computes computation partitions for some statements; this leads to wavefront computation that

P	Execution Time (seconds)						Relative Speedup						Relative Efficiency			
	hand		dHPF		PGI		hand		dHPF		PGI		dHPF		PGI	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
2	-	-	820	–	1213	–	-	-	2.12	-	1.43	-	-	-	-	-
4	436	2094	454	1935	695	2312	4	4	3.84	4.32	2.50	3.62	.96	1.1	.63	.91
8	-	-	259	1086	382	1252	-	-	6.73	7.71	4.56	6.69	-	-	-	-
9	209	783	273	906	381	1296	8.34	10.69	6.38	9.24	4.57	6.46	.76	.86	.55	.77
16	132	466	198	560	222	754	13.21	17.97	8.81	14.95	7.85	11.10	.67	.83	.59	.62
25	88	308	149	389	198	638	19.82	27.19	11.70	21.52	8.81	13.12	.59	.79	.44	.48
32	-	-	127	381	136	508	-	-	13.73	21.98	12.82	16.48	-	-	-	-

Table 2: Comparison between hand-written MPI, dHPF and pghpf for SP (- Numbers are not available because the hand-written code requires a square number of processors, – Numbers are not available because of insufficient per-node memory.)

includes pipelined writebacks of non-owned data. dHPF applies coarse-grain pipelining within the wavefront to reduce the ratio of communication to computation. As with the hand-written versions, the problem size and processor grid organization was compiled into the program separately for each instance with dHPF.

The PGI HPF implementation of SP for pghpf is 4508 lines, 43% larger than the NPB2.3-serial version. The PGI implementation uses a 1D block distribution of the principal 3D arrays along the z spatial dimension for all but the line solve in the z direction. Before the line solve along the z axis, the data for the `rsd` and `u` arrays is copied into new variables that are partitioned along the y spatial dimension instead. This copy operation involves transposing the data. Next, the z sweep is performed locally. Finally, the data is transposed back. The PGI implementation avoids the use of privatizable arrays found in the NAS SP implementations in subroutines `lhsy`, `lhsz`, and `lhsx`. Instead, in the PGI code several loops were carefully rewritten using statement alignment transformations so that intermediate results could be maintained in privatizable scalars instead of privatizable arrays. Applying these transformations to these loops required peeling two leading iterations, two trailing iterations, and writing custom code for each of the peeled iterations. Such transformations are tedious for users to perform. Even more important, in general it is not possible to always eliminate privatizable arrays with alignment transformations. Handling privatizable arrays efficiently in such cases was a key motivation for the computation partitioning for array privatizables. For the PGI code, only a single version of the executable was used for all experiments as per the methodology used for reporting PGI's official NAS parallel benchmark measurements.

Table 2 compares the performance of the hand-written MPI code with the dHPF-generated code and the PGI generated code for both class A and class B problem sizes. For SP, the class A problem is 643, and the class B problem size is 1023. In all cases, the speedups and efficiencies are better for all three implementations on the class B problem sizes. For the larger problem size, the communication overhead becomes less significant in comparison to the computational cost which leads to better scalability. We look to the relative efficiency measures to evaluate how closely the PGI and dHPF-generated codes approximate hand-coded performance. These measures show that the dHPF compiler is generating computational code that is quite efficient since, on 4 processors, it achieves within 4% of the efficiency of the hand-written MPI for the class A problem size, and is 10% better for the class B problem size. On the class A problem size with the PGI compiler, there is a substantial gap between its efficiency and hand-coded performance. However, this gap narrows substantially for the class B problem size. In comparing the performance with dHPF and the PGI compilers, the efficiency of the dHPF-generated code was uniformly better for both class

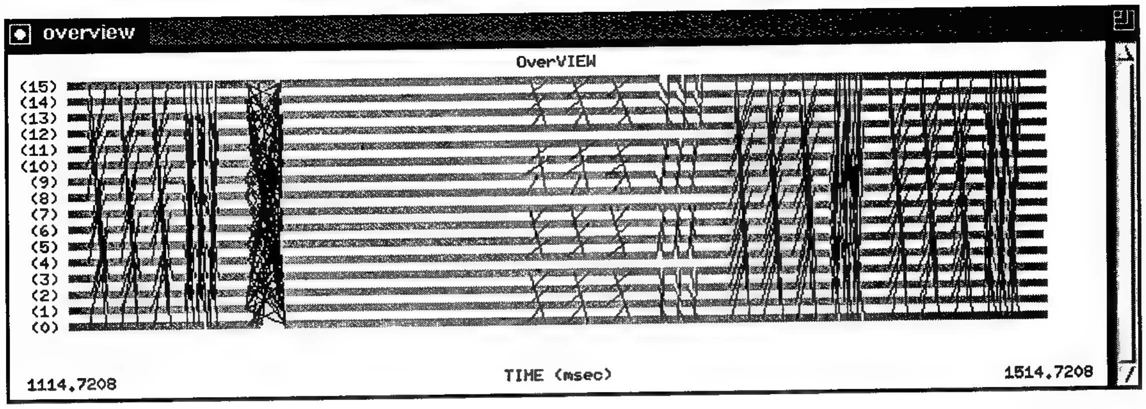


Figure 14: Space-time diagram of hand-coded MPI for SP (16 processors).

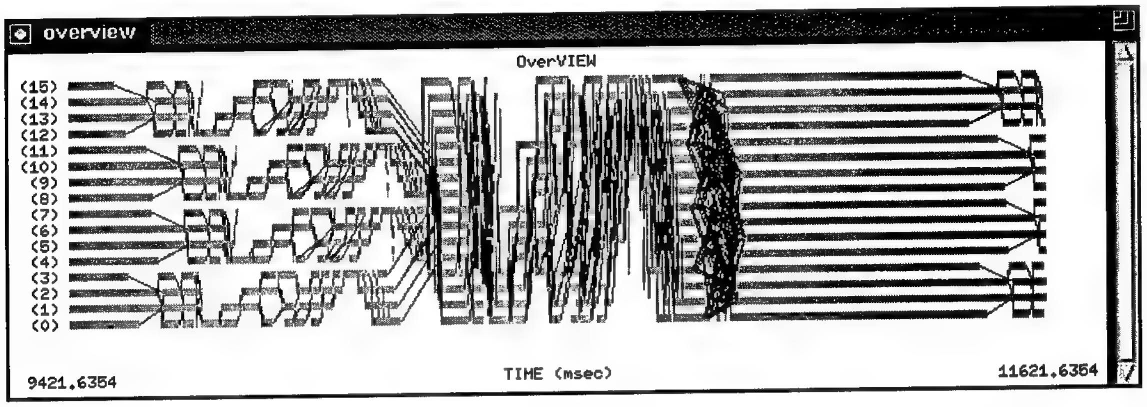


Figure 15: Space-time diagram of dHPF-generated MPI for NAS SP application benchmark (16 processors).

A and class B problem sizes. As the number of processors is scaled for a fixed problem size, the advantages of the multipartitioning in the hand-coded version become more pronounced. For 25 processors, the efficiencies of both of the HPF implementations drops. However, the dHPF code is more efficient by a margin of 15% for the class A problem size and this gap increases to 31% for the class B problem size.

To illustrate the performance benefits provided by multipartitioning in the hand-written MPI code, Figures 14 and 15 show space-time diagrams of 16-processor execution traces from AIMS toolkit [YSM95] for a single timestep of for the hand-coded and dHPF-generated codes respectively. Each row represents a processor's activity. Solid green bars indicate computation. Blue lines show individual messages between processors. White space in a processor's row indicates idle time.

At the left of Figure 14, the first 4 bands of blue correspond to the communication in the `z_solve` phase. The next blue band shows the communication in `copy_faces` which obtains all data needed for `compute_rhs`. The next 4 bands of communication correspond to `x_solve`, and the 4 after that to `y_solve`. The right of the figure shows the `z_solve` phase for the next timestep. Figure 14 shows that the hand-written code has nearly perfect load balance and very low communication overhead using the multi-partitioning.

The leftmost 40% of Figure 15 shows the communication for the `y_solve` phase. The central portion shows the `z_solve` communication. The wide band about 70% across the figure is the communication for `compute_rhs`, which is largely resembles the communication in `copy_faces` in the hand-coded version. The long bands of computation near the right side of the figure belong to compute_rhs and x_solve, which is a totally local computation for the 2D data distribution along the y and z dimensions. Clearly, the largest loss of efficiency is in the wavefront computations of the y_solve and z_solve phases. In each of the phases, there are two forward pipelined computations, and two reverse pipelines. There are several notable inefficiencies evident in this version of the dHPF-generated code. First, it is clear that the pipelines are at different granularities. The leftmost pipeline is quite skewed: processor 0 finishes its work before processor 2 begins, and similarly for each of the groupings of 4 processors. For this pipeline, the granularity is clearly too large, leading to a loss of parallelism. Unfortunately, dHPF currently applies a uniform coarse-grain pipelining granularity to all the loop nests in a program. An independent granularity selection for each loop nest would lead to superior results. Second, between the two forward pipelines in y_solve, communication occurs in the direction opposite the flow of the pipeline which causes a considerable delay before the start-up phase of the second pipeline. This communication is unnecessary and will in the future be eliminated by a less conservative version of our data availability analysis. Overall, however, while the pipeline granularity and performance can be adjusted, it is clear that multipartitioning provides a far better alternative.

3.6.4 BT

The Rice HPF version of BT is derived from the NPB2.3-serial release. Our total changes to the serial code amounted to 226 of 3813 lines which is about 5.9%. The changes we made include:

- Removed array dimension(cache) padding for array u, us, vs, ws, forcing, qs, rho_i, rhs, lhs, square, ainv, and speed. The padding interfered with even distribution of work in the HPF program. Automatic padding of the generated code is performed by the dHPF compiler.
- Eliminated the work_1d common block. Variables cv, cuf, q, ue, buf were instead declared as local variables where needed in the exact_rhs subroutine.
- Added HPF data layout directives to specify a 2D or 3D BLOCK distribution of the common arrays (u, us, vs, ws, fjac, njac, forcing, qs, rho_i, rhs, lhs, square, ainv, speed) in the x, y and z spatial dimensions.
- Added 9 HPF INDEPENDENT NEW directives. In the x_solve_cell, y_solve_cell, and z_solve_cell subroutines, NEW directives were used to introduce two privatizable arrays as temporary variables in two loop nests. In the exact_rhs subroutine, three NEW directives were used to specify cuf,buf,ue,q, and dtemp as privatizable in each of three loop nests.
- In the compute_rhs subroutine, we added an outer one-trip loop along with an INDEPENDENT LOCALIZE directive (a dHPF extension to HPF) for the rho_i, square, qs, us, ws, and vs arrays. This directive has the effect of eliminating communication inside a loop for the specified variables by partially replicating computation of these variables inside the loop so that each element of these arrays will be computed on each processor on which it is used.
- Inlined 3 calls to exact_solution in exact_rhs where our interprocedural computation partitioning analysis was (currently) incapable of identifying that a computation producing a result in a privatizable array should be treated completely parallel.

Performance data for both generated codes by dHPF and PGI, and the hand-written code are shown in Table 3. The format of the table is similar to that of Table 2 for SP. For all three versions,

P	Execution Time (seconds)						Relative Speedup						Relative Efficiency			
	hand		dHPF		PGI		hand		dHPF		PGI		dHPF		PGI	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
4	650	–	609	–	590	–	4	–	4.27	–	4.41	–	1.07	–	1.10	–
8	-	–	322	–	318	–	-	–	8.07	–	8.18	–	-	–	-	–
9	304	–	334	–	315	–	8.56	–	7.79	–	8.26	–	.91	–	.96	–
16	181	715	182	727	171	814	14.33	16	14.28	15.75	15.21	14.06	1.00	.98	1.06	.88
25	117	461	143	534	151	632	22.17	24.85	18.21	21.44	17.25	18.11	.82	.86	.78	.73
27	-	-	137	451	151	503	-	-	18.99	25.40	17.26	22.74	-	-	-	-
32	-	-	108	401	102	508	-	-	24.01	28.54	25.49	22.52	-	-	-	-

Table 3: Comparison between hand-written MPI vs. dHPF and PGI generated code for BT (- Numbers are not available because the hand-written code requires a square number of processors, – Numbers are not available because of insufficient per-node memory.)

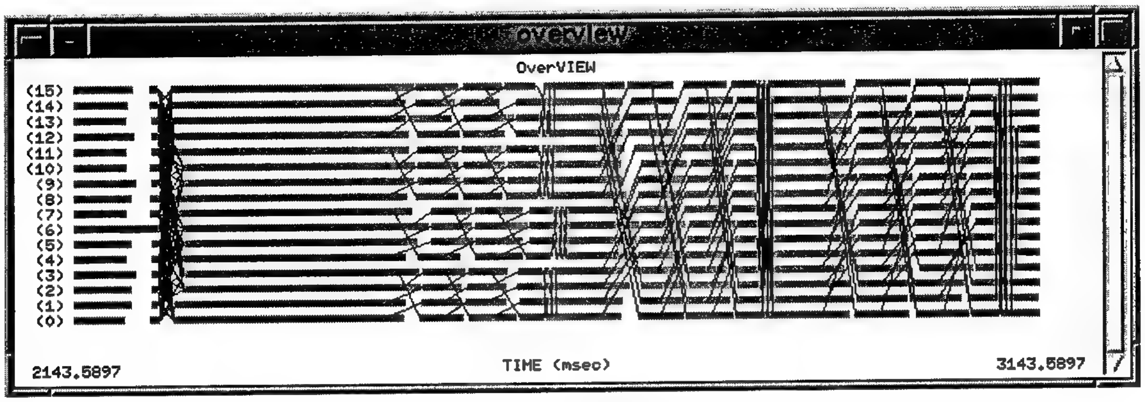


Figure 16: Space-time diagram of hand-coded MPI for NAS BT application benchmark (16 processors).

we include data for both Class A and B problem sizes, which are 64x64x64 and 102x102x102 respectively. All speedups for class A are relative to to the 4-processor hand-tuned version, while those for class B are relative to the 16-processor hand-tuned version. As the data shows, the hand-tuned code demonstrated almost linear speedup up until 25 processors, Our code performs extremely well up until 16 processors for both Class A and B. PGI code has excellent performance for Class A up until 16 processors. Our code outperforms the PGI code for Class B executions and shows better scalability on large number of processors. Efficiency and speedup start to decline from 25 processors for both our code and PGI code because the the wavefront parallelism that we realize for the code using a HPF 2D or 3D block data distribution and the 3D transpose that PGI uses have significant overheads, particularly at higher numbers of processors. As in SP, the sophisticated multipartitioning data distribution strategy used in the hand-tuned code [9] achieves much better scalability, but unfortunately is not expressible in HPF. This can be seen from Figures 16 and 17, which show space-time diagrams of the execution of the hand-written MPI code and the dHPF-generated code for BT respectively. As with SP, the hand-written code shows excellent load-balance and very low communication overhead. The dHPF-generated code is also much more efficient for BT than for SP, but still has significantly higher overheads due to pipelining than the hand-written MPI.

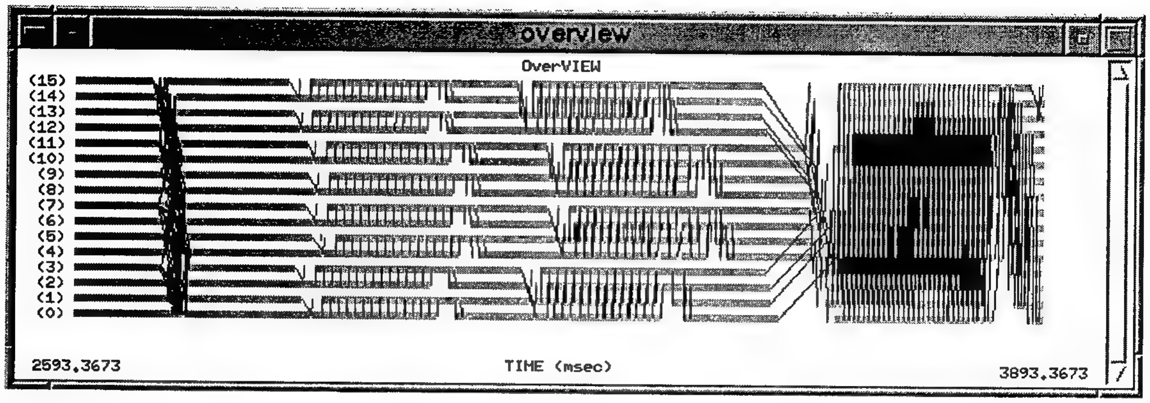


Figure 17: Space-time diagram of dHPF-generated MPI for NAS BT application benchmark (16 processors).

3.7 Compiler-Support for Advanced Partitionings

One important class of tightly-coupled computations not well supported by the HPF and OpenMP partitioning models are those that use line sweeps to solve one-dimensional recurrences along each dimension of a multi-dimensional discretized physical domain. Alternating Direction Implicit (ADI) integration is a common technique for solving partial differential equations that uses this solution style [NNN93]. Two of the NAS parallel benchmarks [BHS+95], SP and BT, use ADI integration to solve the Navier-Stokes equation in three dimensions. Fractional step methods and other solution techniques that use line sweeps are described by Naik et al. [NNN93]. For this class of computations, applying a standard block partitioning to any of the spatial dimensions is problematic—recurrences along the partitioned dimension partially serialize execution.

For the parallelizations of SP and BT as described in the previous section, the dHPF compiler used coarse-grain pipelining. Figure 15 shows a 16-processor execution trace for NAS SP using this approach. For line-sweep computations, block partitionings introduce serialization that coarse-grain pipelining can only partially overcome. Commercial HPF compilers such as PGI's pghpf [BMN+95] even lack support for coarse-grain pipelining. To support the NAS benchmarks, PGI reworked variants of these codes to use full transposes between directional sweeps. Neither coarse-grain pipelining, nor transpose provides ideal scalability.

Hand-coded message-passing versions of the NAS SP and BT benchmarks (version NPB2.3b2) use a sophisticated data distribution known as "multipartitioning" for partitioning multidimensional arrays. Its main properties are that for a sweep along any dimension of an array, (1) all processors are active in each step of the computation, (2) there is perfect load-balance, and (3) the execution requires only coarse-grain communication. Multipartitioning achieves this balance by partitioning data into $p^{\frac{d}{d-1}}$ tiles, where p is the number of processors and d is the number of partitioned array dimensions. Each processor is assigned $p^{\frac{1}{d-1}}$ tiles along diagonals through each of the partitioned dimensions. Figure 18 shows a 3D multipartitioning distribution for 16 processors; the number in each tile represents the processor that owns the block.

For an n-dimensional multipartitioning on p processors, the expression $p^{\frac{1}{n-1}}$ must be an integer. Thus, a 3D multipartitioning requires the number of processors to be a perfect square. However, multipartitioning can be applied to any two dimensions of an n-dimensional array allowing use of an arbitrary number of processors.

Multipartitioning offers two key advantages for parallelizing line sweep computations. First, a multipartitioned distribution of k-dimensional data arrays ensures that for each partitioned di-

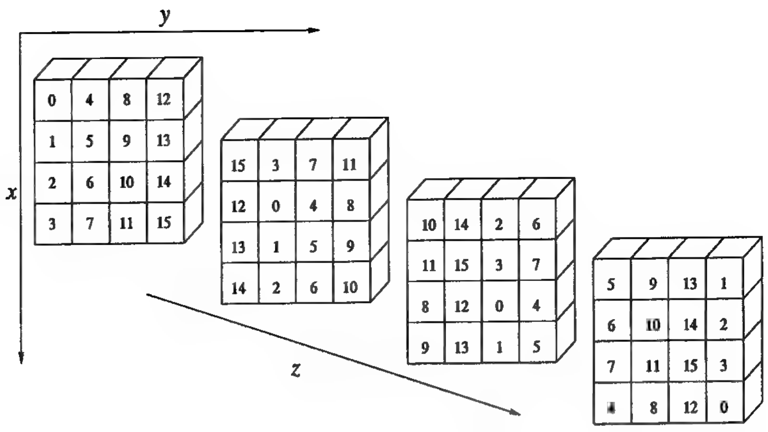


Figure 18: 3D Multipartitioning on 16 processors.

mension, each processor owns a data block in each of the $k-1$ dimensional slabs defined by the partitioning. Computation within a slab is fully parallel, so line sweep computations can be parallelized effectively using this partitioning. Figure 14 shows a 16-processor execution trace that shows the balanced parallelism achieved using multipartitioning for NAS SP. Second, this parallelization only requires coarse-grain communication, unlike pipelining. For these reasons, multipartitioning can provide better scalability and speedup on large systems [Van93].

More generally, many data and computation partitioning problems partitioning a domain into blocks with one or more blocks per processor. We refer to such techniques collectively as *overpartitioning*. Examples in the literature include virtual processor approaches for cyclic and block-cyclic distributions [AL93, GKHS96], support for dynamic and non-uniform computation partitioning of data-parallel programs on heterogeneous systems [NQ93], and support for managing computation on out-of-core arrays [BCK+95].

Here we describe our design and implementation of multipartitioning in the Rice dHPF compiler. This implementation was structured to form the basis for a general overpartitioning framework. We describe several compiler and runtime techniques necessary to generate code for this class of distributions. Our implementation performs the analysis and code generation necessary to realize multipartitioning computation partitionings; however aggregation of messages further communication optimization will be necessary to achieve the scalability of hand-coded implementations.

3.7.1 Multipartitioning in dHPF

To specify that multipartitioning should be used to distribute a multidimensional array, we extended the dHPF compiler to accept the *MULTI* keyword as a distribution specifier.

Because multipartitioning involves distributing each partitioned array dimension across all of the processors, our implementation enforces several restrictions. First, an HPF processor array onto which a multipartitioned distribution is mapped must be a one-dimensional array containing all the processors. Second, if the *MULTI* distribution specifier is used for an array or template, neither *BLOCK* nor *CYCLIC* can be used in the same distribution. Finally, the *MULTI* keyword must be used in at least two dimensions of a distribution.

Virtual Processors The integer set analysis framework used by dHPF [AMC98b] supports *BLOCK* partitionings of arbitrary size onto a symbolic number of processors. To support multipartitioning we extended this model to treat each tile in a multipartitioned array as a block in a

BLOCK partitioned array, mapped to an array of $p^{\frac{d}{d-1}}$ virtual processors.

Implementing this virtual processor model, requires mapping between virtual and physical processors. Each virtual processor is identified by its *tile position indices*, a d-dimensional tuple representing its coordinates in the virtual processor array (in column-major order). We use these indices to index into a *virtual to physical processor mapping.* When a tile needs data from another tile, the other tile's coordinates are computed from the data indices, the virtual to physical processor mapping is then used to determine which physical processor owns the required data.

Memory Model A multipartitioned distribution of an array, requires the allocation of $p^{\frac{1}{d-1}}$ tiles per physical processor. Each tile's data is contiguous. Each tile is extended as necessary with overlap areas [Ger90] to facilitate access to data received from neighboring tiles. On each processor, all local tiles for a multipartitioned array are dynamically allocated as contiguous data. Storage is indexed in column-major order, where the leftmost dimensions are the original array dimensions and a new rightmost dimension corresponds to the *local tile index.*

Code Generation Code generation for multipartitioning is a generalization of code generation for *BLOCK* partitioning. Within dHPF the integer set framework is used to generate code for blocks of arbitrary but uniform size at arbitrary positions. We use the generated computational and communications code for such a block, as the kernel code for each tile of the multipartitioned distribution.

All communication and computation performed for a tile is defined in terms of the data mapped to that tile. Since more than one virtual processor is assigned to each physical processor, the tile position indices have to be adjusted on each physical processor as it cycles through its tiles. Computation for a loop nest occurs by having each processor perform the computations for each of its owned tiles.

To generate code that handles multiple tiles per processor, we wrap a tiling loop that iterates over all the tiles assigned to a physical processor around the kernel code for a tile. The order in which a physical processor must iterate over its tiles, is determined by the data dependences present in the loop nest. Since multipartitioning is a multidimensional distribution, the data dimension in which the outermost dependence is carried is the one that determines the iteration sequence. The order in which each processor iterates over its tiles corresponds to the loop direction and must satisfy the loop-carried dependences present in the loop body. (If there are no loop-carried dependences then the iteration sequence follows the outermost loop index that indexes a multipartitioned dimension within the array.) As shown in Figure 18, the tiles for a processor fall along a diagonal which spans the d dimensions of the array. We use modular arithmetic (modulo number of tiles) to compute the following tile indices from the values of the current ones. The value of the dependence-carrying loop index is what is used to index into the tile dimension of the array. The iteration must begin with the first tile at the appropriate end of the selected dimension.

Communication Model Communication generation for multipartitioned distributions, is a direct extension of the model used to generate communication events for *BLOCK* distributions. For communication loop nests, we applied the same strategy we used for computational loop nests: we extend the basic single tile instance communications loop kernel to support multiple tiles.

For communication that is vectorized outside of all computational loops over data dimensions, we generate a simple tile loop around the communications kernel, applying the same sort of adjustments described for computational loops.

For communication pinned inside a loop by a data dependence, it would be incorrect to wrap the communication event in a tiling loop because dependences would not be satisfied. Code generation for computational loops will have wrapped a tile loop outside the dependence-carrying loop. Thus, no additional tile loop is required for the inner communications kernel code.

In general, a single communication event for a tile may require interaction with multiple communication partners. To manage these interactions, multipartitioning requires a flexible buffering scheme, that supports dynamic allocation of multiple buffers per communication event, for each tile.

Since each physical processor performs communication on behalf of each of its tiles, care must be taken to ensure that messages received are delivered to the appropriate tile. To avoid mixing up messages, we use a message tagging scheme, which uniquely identifies the originating tile and communication event. Each message is labeled with a unique tag consisting of an integer identifying the communication event plus another integer that identifies the originating tile in terms of a unique global number computed from its tile position indices.

Runtime Support Our main runtime components required to support the multipartitioned code generated from the dHPF compiler are a function to compute virtual-to-physical processor mappings and support for managing multiple dynamic buffers. Each multipartitioned template distribution requires a different virtual-to-physical processor map. We associate maps with their corresponding template runtime descriptor. These maps are computed once, at the start of program execution, with very little overhead, since their sizes depend on the number of tiles.

3.7.2 Evaluation of Compiler-Support for Multipartitioning

Here we briefly describe a preliminary evaluation of prototype multipartitioning support in dHPF using the NAS SP application benchmark to compare a hand-coded MPI implementation using multipartitioning to dHPF-generated code that uses multipartitioning to parallelize a serial version of the benchmark to which data layout directives have been added.

Our scalability experiments were performed on an SGI Origin 2000 node of ASCI Blue Mountain (128 250MHz R10000, 32KB (I)/32KB (D) L1, 4MB L2 (unified)) using 1, 4, 9, 16 and 25 processors for each execution instance.

From a compiler's point of view, the NAS SP benchmark presents a significant challenge for achieving high performance. To generate efficient parallel code from a lightly modified serial version of this benchmark with the BLOCK distribution, requires the use of several advanced compilation strategies [AJMCY98] including non-owner-computes computation partitionings, complex patterns of computation replication to reduce communication for privatizable arrays and other loop independent data reuse, aggressive communication coalescing, coarse grain pipelining, and interprocedural computation partitioning. These optimizations together yield reasonably good performance even with BLOCK partitioning, within about 20% of the handcoded message-passing version on 32 processors, where the number of processors is a known compile-time constant. The scalability of the compiler-generated code using coarse-grain pipelining (as described in the previous section) falls substantially short of the handcoded version because of inherent serialization induced by the *BLOCK* partitioning.

Figure 19 shows a 16-processor parallel execution trace for one iteration in the steady-state section of the NAS SP class 'A' execution, for the multipartitioned code generated by the dHPF compiler. By comparing this execution trace with that of the hand-coded MPI version shown in Figure 14, one can see that dHPF-generated code achieves the same qualitative parallelization.

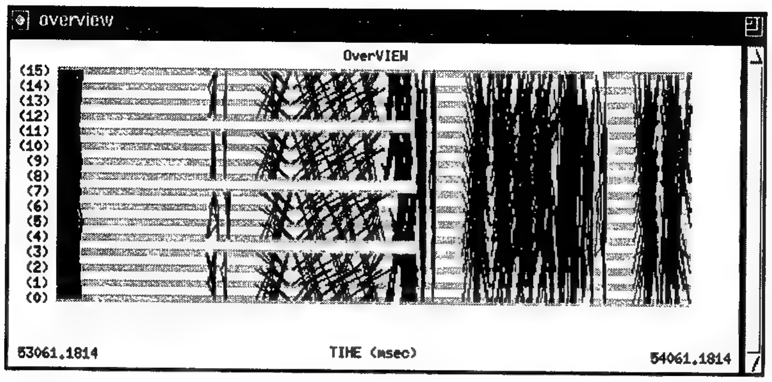


Figure 19: dHPF-generated NAS SP using 3D multi-partitioning.

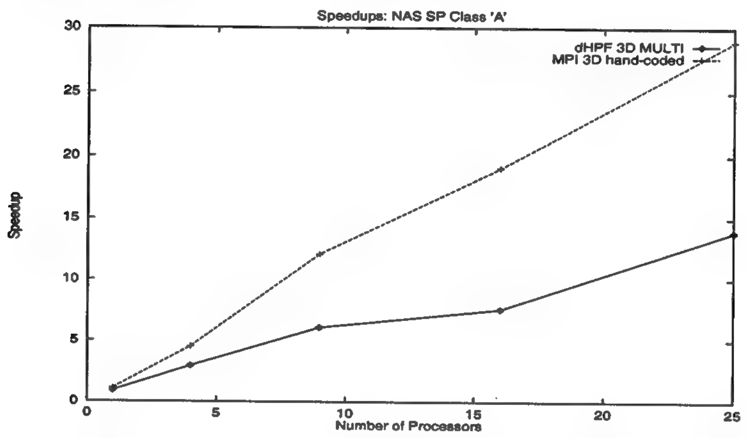


Figure 20: Speedups for MPI hand-coded multipartitioning and dHPF-generated multipartitioning versions of NAS SP benchmark (class A).

Comparing this multipartitioned version with the coarse-grain pipelining code for a BLOCK distribution shown in Figure 15, it is clear that the new compiler-generated multipartitioning code has less serialization.

Despite the fact that the dynamic communication patterns of our compiler-generated parallelization using multipartitioning resemble those of the hand-coded parallelization, there is still an important performance gap between our dHPF-generated code and the hand-coded MPI. Figure 20 shows speedup measurements taken in June 2000 on an SGI Origin 2000 system equipped with MIPS R10000 processors. All speedups shown in the figure are relative to the performance of the sequential code for SP from the NAS 2.3-serial distribution.

The most significant difference between the performance of the codes is that the dHPF-generated code does not yet have the same scalability as the hand-generated code. The performance gap is primarily due to insufficient aggregation of communication. First, in the dHPF-generated code, communication that has been fully vectorized outside all loops over spatial dimensions is performed by each processor one tile at a time rather than once for all tiles. For 3D multipartitioning, when shifting array values along a spatial dimension this effect causes $O(p^{\frac{1}{2}})$ messages instead of the $O(1)$ messages in the hand-coded multipartitioning. Enhancements to dHPF's communication generation and run-time libraries are underway to enable messages to be sent on a per-processor rather than a per-tile basis. Second, in the dHPF-generated code, separate messages are used to

Metric	Hand-MPI	dHPF-MPI
cycles	.94	1.22
grad. instr.	.94	1.07
grad. loads.	.92	.96
L1 misses	.95	.98
L2 misses	.94	1.04
prefetches	1.02	.08

Table 4: Ratio of performance metrics for single-processor executions of parallelized versions of the NAS SP benchmark relative to those for the original sequential code.

move data for each array that must be communicated. Such messages should be coalesced. Finally, communication for references involved in carried data dependences along a partitioned dimension may not be fully fused when the references require communication in different sets of iterations.

In January 2000, first detailed measurements of the scalar performance of the dHPF-generated code for multipartitionings showed that its scalar performance was a factor of 2.5 slower than the original sequential code. Careful analysis of overhead in the generated code showed that the main contributing factors were high primary data cache miss rates, excessive instruction counts due to complex addressing using linearized subscripts and overly complicated communication code, and excessive code replication.

To address these issues, we developed a number of compiler refinements including communication hoisting (which makes it possible to nest loops according to their natural memory order rather than constraining communication-carrying loops to be outermost), array padding for dynamic arrays to reduce cache conflict misses, data indexing using Cray pointers rather than linearized storage to help the back-end compiler optimize array subscript calculations, communication set splitting to avoid complex code that comes from having a single communication event orchestrate data movement across multiple partitioned dimensions, and code generation for multiple loop nests at a time to reduce code replication that can arise from guard lifting.

Table 4 uses several metrics to compare the performance of hand-coded MPI, dHPF-generated MPI (as of May 2000), and the original sequential code. These measurements were collected on a single node of an SGI Origin 2000 equipped with a 300 MHz MIPS R12000 processor. All code was compiled using the SGI Fortran 77 compiler version 7.3.1.1.[10]

The overall scalar performance of the dHPF-generated code is competitive with both the hand-coded MPI and the original sequential code. The number of graduated instructions, graduated loads, and cache misses measured for the dHPF-generated code are within 4–7% of the values measured for the sequential code. However, the overall performance of the dHPF-generated code was 22% slower than the sequential code. The primary contributing factor was that the SGI compiler failed to generate data prefetches for the dynamic arrays in the dHPF-generated code.

At this point we are optimistic about achieving performance and scalability very competitive with hand-coded multipartitioning, once ongoing work on communication optimizations is completed.

[10]The compiler flags used were `-64 -r12000 -OPT:Olimit=0 -NC200 -OPT:alias=cray_pointer -LNO:prefetch=2 -O3` in order to attain high performance

3.8 Using Data-Parallel Languages for Irregular Applications

Irregular applications contain references for which a closed form representation of the data accessed cannot be computed statically. On message-passing parallel systems, these applications rely on runtime libraries to identify accesses to off-processor data and to coordinate data movement. To keep the overhead of this approach manageable, a good data distribution and communication aggregation are very important. While HPF generalized array assignments using *forall* and *scatter* statements support irregular data movement, the appropriateness and efficiency of such constructs is still largely unproven for full-scale irregular applications.

As part of the research in the dHPF project, we performed a comparative study of several implementations of an irregular application for n-body simulation. All implementations use an adaptive version of Anderson's method for hierarchical approximation of far-field interactions [And92]. Hierarchical methods for n-body simulation have been of interest to the computational science community not only because of their speed and accuracy, but also because their irregular structure makes efficient parallelization difficult. Hu and Johnsson developed an HPF implementation of an adaptive hierarchical solver using Anderson's method that served as the basis for much of our work [HJ96, HJT97]. Their landmark implementation demonstrated that sophisticated algorithms for highly irregular problems *can* in fact be implemented in HPF. However, a performance comparison of their HPF implementation with a hand-coded, MPI-based parallel implementation that we developed exposed some costly inefficiencies in the HPF implementation that slow it's running time by as much as a factor of three.

Our comparative study made several contributions:

- We constructed a sophisticated MPI implementation of an adaptive version of Anderson's method which integrates proven techniques to achieve good performance and scalability.
- We performed a careful measurement and characterization of overhead in Hu and Johnsson's HPF implementation relative to our MPI reference implementation.
- We described a modification to Hu and Johnsson's communication strategy that, when integrated into their implementation, eliminates more than 75% of the performance difference relative to the MPI implementation.

A detailed description of this study is described in a publication [MMC99].

The next section briefly describes hierarchical n-body methods to provide a context for understanding comparisons of HPF and MPI implementations of Anderson's method. Then, we describe the key source of inefficiency in Hu and Johnsson's HPF implementation that we identified in comparison with a hand-coded MPI implementation we developed. Finally, we describe a modification to the HPF approach that dramatically improves performance.

3.8.1 Hierarchical Methods

To compute far-field forces rapidly, hierarchical methods aggregate the effects from bodies a sufficient distance away, computing their influence as part of a group, rather than individually. The principal data structure used to construct groupings for these methods in 3D is an oct-tree. One constructs an oct-tree starting with a root box that contains all of the bodies and then recursively subdividing boxes into 8 boxes of equal size until a stopping condition is met.

Once the tree is formed, an upward pass over the boxes in the tree establishes the far-field approximations for each box. At the leaves, the approximation for a box is computed from the bodies within; at higher levels, the approximation for a box is computed from approximations for boxes it contains. The form of the approximation is application specific.

Here we describe a progression of three $O(n)$ hierarchical methods. First we introduce Greengard and Rokhlin's fast multipole method (FMM) [GR87]. Next, we describe the adaptive variant of this algorithm. Finally, we describe Anderson's method, which has the same algorithmic structure as the FMM methods, but but a different numerical technique for approximating far-field forces.

Fast Multipole Method. Rather than computing far-field for each body individually, as in the $O(n \log n)$ Barnes-Hut algorithm [BH86], Greengard and Rokhin's Fast Multipole Method [GR87] makes use of the observation that when a box A and a box B are "well-separated", the far-field effect of the bodies in box B on those in box A, and vice-versa, can be approximated as a single interaction between the *boxes*. Such interactions between well-separated boxes occur at all levels of the tree, and the savings in computation enable FMM to compute far-field forces in $O(n)$ time. Interactions are computed in a downward pass over the tree. At each level, interactions are computed between boxes at that level that are well-separated, and the results, collected in the form of a "local-field potential," are passed down to the next level. At the lowest level, the local-field potential for a box is passed down to each body inside and interactions between bodies not sufficiently separated are computed.

Adaptive FMM. The FMM algorithm just described assumes a tree of uniform depth. An adaptive variant avoids unnecessary refinement by not subdividing any box that contains fewer bodies than a specified threshhold. The key difference with respect to the non-adaptive algorithm is that the set of boxes with which a given box will interact is not statically known and must be computed from the shape of the adaptive tree by a somewhat complicated algorithm. To simplify implementation and maximize cache locality, several types of interaction lists for each box are computed before the upward pass and a separate computation phase is added between the upward and downward passes. There are three types of interaction computations and therefore three lists: boxes in *list1* are adjacent leaf boxes (and therefore are not sufficiently distant from each other to allow approximation); boxes in *list2* are the same size and well-separated (that is, sufficiently distant from each other to allow approximation); finally, boxes in *list34* are different sized and well-separated from the perspective of one of the boxes but not the other.

Anderson's Method. The algorithmic structure of Anderson's method [And92] is the same as that of FMM. Its key difference from FMM is in the way it propagates potentials. For three-dimensional problems, the computational element of FMM is a multipole expansion located at the center of an abstract sphere containing the cluster of bodies; in contrast, Anderson's approximation computes potentials at locations on the circumference of a sphere. Compared to multipole methods, Anderson's method achieves the same level of accuracy with fewer levels in the tree.

3.9 Implementation Comparison

Here, we first describe our hand-coded implementation of Anderson's method. Next, we describe highlights of Hu and Johnsson's HPF implementation. Finally, we describe a comparison of the performance of these implementations for several problem sizes and processor counts on a Cray T3E. This comparison shows that the HPF implementation has some significant inefficiencies relative to the hand-coded one.

3.9.1 The Hand-coded Implementation

The FMM program in the SPLASH-2 suite from Stanford [WOT$^+$95] was the starting point for development of our hand-coded MPI implementation of Anderson's method, though our implementation now bears little resemblance to the original. Among the structural changes we have made to the code:

- We use MPI-based explicit communication rather than shared memory.
- We replaced the multipole expansions with Anderson's method for computing potentials.
- We use a 3D octree as the basis for the hierarchical solver rather than a quadtree.

The principal remaining similarity between the implementations is in the record structures used by the hierarchical solver. Below we describe key features of our MPI implementation.

Body Distribution. To distribute bodies among processors, we first compute the position of each body along a Hilbert curve[11], and then sort the bodies according to their position along the curve. Since both the Hilbert curve and the octtree recursively divide space in half along each dimension, all bodies in the same leaf of the octtree are contiguous after the sort. Next, we partition the sorted sequence of bodies among the processors by assigning each processor a contiguous range. We select the partition points to ensure that each processor is assigned all bodies in a subtree of the octtree. With this partitioning, we are able to construct octtrees locally, except for a brief communication phase in which processors exchange information about *shared boxes* (boxes at upper levels of the tree whose subboxes lie on more than one processor) to ensure that the representation of these boxes is globally consistent.

Construction of Neighbor Lists. A key step in adaptive hierarchical methods is building the interaction lists for each box, as described in Section 3.8.1. The fine-grained nature of the computation in this phase, combined with its large communication requirements, causes it to be a major bottleneck in the parallelized application if special care is not taken. By transforming the uniprocessor list construction algorithm into a form that enables us to gather non-local data using an efficient inspector-executor strategy, we are able to dramatically reduce the impact of list construction on the parallel runtime. (Details of our list construction algorithm are described in [McC99].)

Propagation of Potential Information. As noted before, we replicate information about shared nodes at the uppermost levels of the tree to all processors and ensure that all nodes in a subtree below any non-shared node are located on the same processor. This partitioning strategy avoids communication in the downward pass and requires only a single communication step in the upward pass when moving from private nodes to the shared parents.

Interaction Computation. As in list construction, we communicate non-local data required in the interaction computation using a variation on the inspector-executor technique. Computation is divided into 3 parts: list1 interactions, list2 interactions and list34 interactions. To ensure load-balance, we move data for boxes involved in each of the three computation phases into a

[11]Hilbert curves [Sag94] are one of a class of continuous, non-smooth, "space-filling curves" that map a 1-dimensional interval to an N-dimensional volume. Such curves can be constructed to pass arbitrarily close to every point in the volume.

"weighted-block" distribution immediately prior to that phase. This involves looking in the work-list for each box involved in the computation to determine the amount of work it will do, and then minimally redistributing the boxes such that each processor will have approximately the same total amount of work.

3.9.2 The HPF Implementation

Details of Hu and Johnsson's implementation of Anderson's method can be found in [HJT97]. Here we provide only a brief overview of some similarities and differences between their implementation and our hand-coded MPI implementation.

- They represent objects using multiple attribute arrays rather than a single record structure.
- They express communication of non-local data for irregular references using generalized array assignments to "gather" the data before computation. A published paper describing this work discusses this issue in more detail [MMC99].
- They distribute bodies using a space-filling curve, though they do not exploit the relationship with octtrees to minimize communication during the tree-building phase. Instead they construct the tree level-by-level, block distributing the data for each level. As a result, parent data is *not* necessarily on the same processor as child data.
- They have parallelized the uniprocessor list creation algorithm in a fashion that requires more communication rounds than our approach.
- As a result of their level-by-level block distribution of boxes, they must communicate between each level during the upward and downward passes for propagating potentials.
- They use a clever scheme that allows enables them to approximate a weighted block distribution to load balance the interaction computations.
- They have an extra repartitioning phase after list construction. This step moves box data into the weighted block distribution described above.

3.9.3 Initial HPF vs. MPI Performance

Here we summarize the results of a performance comparison of three application variants performed on a Cray T3E-600 at the San Diego Supercomputer Center. We compared Hu and Johnsson's original HPF implementation, our hand-coded MPI implementation, and a variant of the Hu and Johnsson's HPF implementation that, for some phases of the algorithm, uses calls to specialized communication routines Hu and Johnsson developed. The HPF programs were compiled using PGI's PGHPF compiler, release 2.3-1 for the T3E.

In our experiments, we scaled problem size with the number of processors, so a doubling of processors implies a doubling of bodies simulated. Bodies were initially distributed according to a Plummer distribution [AHW94]. Each application variant was run for a single timestep on 8, 16, 32, and 64 processors. Those that didn't run out of memory were run on 128 processors.

The most significant differences in partitioning strategy between the HPF and MPI implementations are for the tree construction, upward pass, and downward pass. In these phases, the hand-coded implementation uses a more efficient strategy based on replication of shared nodes in the upper levels of the tree. However, despite these differences, these phases show the smallest differences in performance when comparing the HPF and MPI implementations. The hand-coded implementation always has less overhead; however, the difference in overhead is fairly consistent as the problem size gets larger and the number of processors increases. For example, this overhead is 3-5% for the upward pass.

The remaining phases, which differ the least in terms of partitioning strategy, have the most significant differences in performance when comparing the HPF and MPI implementations. The largest differences are for interactions between well-separated boxes. For 64 processors the overhead of the HPF implementation is a *factor of 84* greater than that of the hand-coded implementation. This difference translates to a substantial difference in running time: the HPF implementation took longer to compute interactions between well-separated boxes than the hand-coded version took to complete an entire timestep cycle.

3.9.4 Rationale for Performance Differences

The performance comparison described in the previous section found that the HPF implementations were substantially less efficient than the hand-coded MPI implementation. Our analysis of the applications showed that the most significant performance differences were due to how the different application variants satisfied (recognized and communicated for) references to off-processor data.

Since no closed form representation of the data accessed by an irregular reference can be computed statically, runtime processing is needed (a) to determine which (if any) accesses through an irregular reference will access off-processor data, and (b) to coordinate necessary data movement. Rather than performing communication separately for each access to non-local data, it is advantageous to first determine the locality for all dynamic instances of an irregular reference in a loop and then communicate for all non-local values in a single step. This strategy is known as the "inspector-executor" paradigm [MV89, SCMB90]. The *inspector* determines what non-local data will be accessed, and the *executor* performs the computation on local data and localized non-local data. Both the HPF and MPI implementations use variations of the *inspector executor* paradigm to acquire non-local data needed to satisfy references.

Hand-coded Inspection Our hand-coded MPI implementation uses a form of inspector-executors adapted to our choice of Warren and Salmon "hashed-octtrees" [WS93] for the octtree data structure of the application. Looking up a node in such trees uses a unique identifier (representing the nodes location in the tree) as a key for accessing nodes in a hashtable representing the tree. This approach simplifies management of distributed trees in two ways. First, the identifier for a node is the same on *all* processors. Second, integration of non-local nodes into a local tree is simple: data for non-local nodes is simply added to a processor's hashtable.

Here we describe how we gather non-local tree nodes needed by a processor. First, each processor inspects its local portion of the computation for accesses to non-local nodes and collects identifiers for these nodes into a hashtable. Next, all processors exchange the IDs in their off-processor hashtables. Third, each processor searches its tree for data requested and then replies with the necessary data if found. Finally, each processor inserts non-local data received into its tree hashtable and the loop computation continues without further interruption.

This inspection phase enables each processor's computations at nodes in the octree to look up information necessary information about interacting nodes (including non-local nodes) without any regard to where the principal copy of the data resides. The inspector phase ensures that all processors have copies of any nodes they need. During the executor phase, each processor continues to access data using an irregular access pattern (in this case, a hash table lookup).

Inspection in HPF Hu and Johnsson's HPF implementation uses generalized array assignments to gather non-local data in what amounts to a variation of the inspector-executor technique. (A more detailed description of this issue can be found in a published version of this work [MMC99].) Their approach has the effect of converting potentially non-local irregular references in the executor into local regular references. In the HPF language, regularization is the standard idiom for handling irregular references. This regularization strategy differs qualitatively from the inspection approach used in the hand-coded application which enables references in the executor to remain irregular. This difference accounts for the principal differences in efficiency between the HPF and hand-coded MPI implementations.

Regularization appears to be a simple and elegant solution for handling irregular references in HPF. However, transforming irregular references in the executor into regular references results requires replication of a substantial amount of data for this application. For an illustration of the potential impact in our implementation of Anderson's method, consider the interactions between well-separated boxes. Suppose box B is an interior node in a full tree; it is then well-separated from 189 other boxes in a 3D oct-tree. Each of those boxes is also well-separated from B. Thus, B appears on interaction lists of its 189 neighbors. Using the regularization strategy to localize references to B in those lists thus causes B's data to be copied 189 times, potentially across processor boundaries.

There are two ways in which such copying can hurt performance. First, multiple copies of B are potentially communicated to a single processor (if B appears on multiple lists on that processor), thereby increasing communication volume and latency. Second, whether B's data is local or nonlocal, new space must be allocated to store it, resulting in as much as a factor of 189 difference in B's storage requirements (if B is only on local lists).

Our analysis indicates that the additional storage required for regularization is ultimately responsible for the performance difference between the HPF implementation and its hand-coded counterpart. Although the runtime processing required to implement inspection should ideally be placed outside of as many loops as dependences will allow, resource constraints might force their placement within loops. In several phases of the Hu and Johnsson's HPF implementation, their regularization strategy increases storage requirements to such an extent that they needed to place their gather/inspection code within some loops with many iterations. In contrast, we are able to place these inspectors at the outermost level in our hand-coded MPI implementation because we use a hashtable to eliminate storage redundancy.

3.9.5 Improving HPF Performance

To improve HPF performance, we must avoid the redundancy associated with regularization. Unfortunately, there are few alternative to regularization. In the HPF language proper, we have found that *programmer*-driven inspection of data for locality and redundancy is difficult or impossible because:

- there is no straightforward means of accessing data owned by a particular processor, and
- there is no notion of arrays local to a processor.

We know of no method that a programmer can use programmer to circumvent the redundancy inherent in the regularization strategy entirely within the HPF language. We *have* found, however, that a modest use of HPF "extrinsic procedures" [Hig93, KLS+94] enables us to avoid the redundancy while retaining the benefits of the high-level HPF model for the rest of the application.

HPF extrinsic procedures enable HPF programs to call code not written in strict data-parallel style and allow the called code to operate on distributed arrays defined in the HPF program.

HPF_LOCAL extrinsic procedures, in particular, have at their disposal several intrinsic functions that are unavailable at the global level. For example, the intrinsic function `size` can be used within an HPF_LOCAL extrinsic to determine the *local* extent of a dimension of a distributed array passed as an argument. Because extrinsic procedures enable processors to perform different operations in parallel (as opposed to identical operations on different parts of the same array) their use is frowned upon by data-parallel purists.

We are interested solely in the ability extrinsic procedures give us to define local arrays and manipulate local sections of distributed arrays. We use this capability first to locally collect a vector of unique indices that will be used to indirectly access a distributed array and then later to perform a communication-free computation with localized data.

We applied this inspection strategy using HPF_LOCAL to the phases of Hu and Johnsson's HPF implementation where our performance comparison indicated that they were needed most: list construction, and interaction computation. This approach dramatically improved performance of the HPF code. Our experiments showed that using an HPF_LOCAL inspector reduced overhead in the list creation phase by roughly a factor of 10. When this strategy was used in conjunction with Hu and Johnsson's MPI-based collective communication rather than PGHPF's default communication, the overhead dropped by an additional factor of two or more.

Only the interactions between boxes that are different sized and well-separated from the perspective of one of the boxes but not the other are somewhat problematic using the HPF_LOCAL approach. We attributed this to the fact that we are forced to use a rather inefficient communication method to move the boxes into the weighted-block distribution. Specifically, because of the way data is laid out and the formulation of the function which determines the new distribution, we are forced to use a `scatter_copy` rather than a `forall` to move the data; unfortunately `scatter_copy` does not provide a good way to move multi-dimensional data. One is forced to use a very general technique with which it is possible to describe the new position of every single element in the array; in this case, we simply want all elements of a row to follow the first element.

3.9.6 Discussion

We have described and evaluated two sophisticated parallel implementations of an adaptive, hierarchical solver which uses Anderson's method for calculating interactions in n-body systems: our explicitly-parallel MPI implementation and Hu and Johnsson's data-parallel HPF implementation. Our measurements of these implementations demonstrated that there were significant performance differences between the hand-coded MPI and the HPF implementations. We determined that the primary source of inefficiency in the HPF implementation was redundant communication that was necessary to initialize redundant storage that is used for regularizing indirect references. We demonstrated that when this redundancy is eliminated in just two phases of the HPF implementation, with the aid of HPF_LOCAL semantics, the performance of the otherwise unchanged HPF implementation much more closely approaches that of the hand-coded version.

On 64 processors, while the original HPF implementation incurs nearly 300% more overhead than the hand-coded version, our revised HPF implementation reduces the gap to just 50%. If we accept the assertion that collective communication support in the PGI HPF compiler could be improved to approach the efficiency of the specialized communication routines written by Hu and Johnsson (used by the HPF-MPI and HPFLOC-MPI versions) then there is a fairly consistent gap of about 25% left between the performance of the hand-coded MPI implementation and the revised HPF implementation.

Two questions remain. First, what accounts for the remaining performance gap and can it be bridged? Second, what are the implications of this work for irregular computation and HPF?

Remaining Gap. Almost all of the remaining difference in running times between the two implementations can be attributed to the phases of the algorithm that we did not change. The one exception would appear to be the interactions between well-separated boxes. However, 80% of the difference in time for this phase is due to a small difference in the *computational* algorithm: the HPF versions have a test in an innermost loop which we avoided in the MPI implementation. If we discount this algorithmic inefficiency, we find that the phases of the HPF implementation that we didn't change account for 86% of the remaining performance gap while the phases we changed account for only 14% of the remaining gap.

There are several factors which contribute to the remaining 14% performance gap in the application phases we modified to use the HPF_LOCAL-based inspection strategy. First, we took great pains in the MPI implementation to ensure that no communication of data "holes" takes place, whereas the HPF implementation cannot. For example, when communicating the particles associated with a leaf box, we communicate and store exactly the number of particles associated with that box, while the HPF implementation communicates and stores the maximum number of particles per box for every box. Second, our use of structures to group data in the MPI implementation results in a single communication per object whereas the HPF implementation's use of attribute arrays results in multiple communications per object, which increases communication overhead.

These same factors, of course, also contribute to the performance gap for the phases that we did not modify. There are two components to the remaining difference in the unmodified phases: the extra partitioning phase required in the HPF application, and the algorithmic phases (upward and downward passes, tree build). We first note that the extra partitioning phase was not improved by the use of Hu and Johnsson's MPI routines, primarily because it uses more complicated communication constructs such as scan reductions that they chose not to implement in MPI. The time for this phase could potentially be lower if the PGI runtime library communication routines can be improved.

After accounting for the factors discussed above, we attribute the rest of the performance differences to differences in partitioning strategies used by the HPF and MPI implementations. The main feature that distinguishes the partitioning strategy we used in MPI from that used in HPF is the notion of processors sharing boxes in the upper levels of the tree. Given the restrictions on user knowledge of processor/data relationships imposed by the HPF language, it is not clear how one could implement this strategy in HPF. Perhaps instead of a single array representing boxes, one could split them into two sets: those owned by a single processor in a distributed array and those owned by all processors in a separate replicated array. Whether a compiler could efficiently coordinate data motion between the two sets as we have in our implementation, is not obvious.

Implications. We have shown that if arbitrary irregular applications are to be implemented with high efficiency in data-parallel languages such as HPF, then special care must be taken to avoid redundancy in the communication and storage of non-local data. We believe that this work therefore has implications for two groups:

1. *Vendors* of HPF compilers need to provide better support for automatically generating appropriate inspector-executor code for irregular references. Techniques described by von Hanxleden [Han94] in his dissertation would suffice for the cases we encountered, though extensions to these techniques may be needed for more complex cases, such as those involving multiple levels of indirection. Without compiler-synthesized inspectors and executors, we know of no way to eliminate the communication redundancy that results from regularization without dropping into HPF_LOCAL extrinsics. Attempts to achieve the desired effect by not using a forall loop for regularization of irregular reference patterns caused the PGI HPF compiler to

employ run-time resolution with which no speedup is possible.

2. Until better inspector-executor support becomes widely available in HPF compilers, HPF *application developers* would do well to follow our example and use HPF_LOCAL extrinsic procedures to implement inspector-executor style handling for irregular references in cases where storage and communication redundancy prove significant.

3.10 Compiler and Run-time Support for Software Distributed Shared-Memory

Software distributed shared-memory systems implement shared-memory communication abstractions on top of message-passing systems, thus providing a flexible base for developing parallel applications. Shared memory is particularly suited to irregular applications because its dynamic resolution of communication greatly simplifies access to, and management of, irregularly shared data. However, the types of regular applications supported by SDSMs are limited. The regular, data-parallel applications used in the previous studies [LCD+96, DCZ96, HTK98, CL97] all had their computations partitioned along the single, slowest varying dimension of the principal array. Because SDSMs use blocks that are multiples of system page sizes, partitioning along other dimensions causes a large number of pages to become fragmented, containing only a few truly shared values along with a large amount falsely shared data.Such fragmentation results in disastrously high communication costs. However, applications with directional sweeps across several data dimensions, such as those common in computational fluid dynamics codes (e.g. [BHS+95]), can reduce serialization by partitioning computation in multiple dimensions. Multi-dimensional partitionings are also more scalable than one-dimensional partitionings because they results in a lower communication/computation ratio. For these reasons, supporting multi-dimensional partitionings effectively on an SDSM is an important problem. Regular applications in previous studies exhibit loosely-synchronous parallelism, namely, completely parallel loop nests that are separated by synchronization. Noticeably absent are regular applications that require more tightly-coupled synchronization such as wavefront parallelizations of applications with loop-carried data dependences.

As part of the dHPF project, we devised and evaluated techniques for compiling High Performance Fortran(HPF) to page-based software distributed shared memory systems(SDSM). Our work focused on reducing the cost of false sharing[12] and fragmentation[13] caused by multi-dimensional partitioning. We exploit compiler-derived knowledge of sharing and communication patterns to help choreograph SDSM synchronization and data movement. In addition, we leverage compiler knowledge of synchronization patterns by extending DSM synchronization mechanisms to support pairwise synchronization and to support reductions efficiently using an extension of the barrier implementation. Although pairwise synchronization usually reduces the number of synchronization messages, more importantly, it enables us to organize wavefront computation. The combination of techniques we describe efficiently and effectively supports not only loosely-synchronous parallelism with multi-dimensional computation partitioning but also tightly-coupled applications.

In our work, we developed two novel techniques. The first technique, the *compiler-managed restricted consistency*, use compiler-derived knowledge to delay the application of memory consistency operations to data that is provably not shared in the current synchronization interval, reducing false sharing. The second technique, the *compiler-managed shared buffers*, when combined with restricted consistency, eliminates fragmentation. The two techniques can efficiently

[12]False sharing occurs when two or more processors access disjoint sets of data elements in the same block.
[13]Fragmentation occurs when blocks are communicated for only a small fraction of data.

support multi-dimensional computation partitioning and parallelization of wavefront computation on SDSM. We explain these techniques briefly below.

Compiler-Managed Restricted Consistency. Whenever processors synchronize, the multiple-writers protocol used in TreadMarks will re-establish the consistency of falsely-shared pages, even if they will continue to be falsely-shared. To avoid such unnecessary consistency maintenance, we further weaken the standard lazy release consistency, multiple-writer model [ACD+96] by having a `signal` operation create consistency meta-data *only* for modified pages that *might* be accessed by its synchronization partner. In general, the compiler proves that some set of data pages are guaranteed not to be shared between this synchronization operation and the next, and enforces consistency only for pages outside this set. For regular applications, the dHPF compiler computes precisely the set of pages that must be communicated.

For irregular applications, our analysis is inexact but conservative. The barrier or lock calls are provided similarly with such a set of pages that either the compiler or the programmer can prove not to be shared in the current synchronization interval. The compiler may peel or split a loop when the synchronization intervals in different iterations involve different set of pages to synchronize. See [Zha99] for detail.

Compiler-Managed Shared Buffers. As noted earlier, multi-dimensional computation partitionings can cause substantial fragmentation and false sharing. When pages contain only a few shared values, we marshal the actively shared data into separate out-of-band, shared buffers, and transform the read reference to access the data in the shared buffer. The compiler algorithm that detects such data is testing whether the array access is stride-one in a loop. To minimize false sharing on the buffer, we allocate a new page aligned buffer for every pair of destination processor and array. Moving the actively shared data to a compact set of densely packed pages enables restricted consistency to skip consistency operations on the fragmented pages. The computational overhead of using compiler-managed shared buffers includes the cost of splitting loops and data copying, which is relatively small.

Shared-buffer also applies to irregular applications. Take molecule dynamic and fluid dynamic applications as examples. These applications involve indirect access of shared arrays through usually an interaction list. Through this list, each processor usually has a fixed set of elements to access and modify for a number of iterations. False sharing and fragmentation may arise if the index set is scattered on the shared arrays. Shared buffers can be allocated to hold only those elements from the index set. Each processor will access the buffer of its own interest. This requires that the index sets are precomputed by a preprocessing loop from the interaction list.

DSM Enhancements. To coordinate the pairwise sharing and enforce dataflow constraints efficiently on the TreadMarks SDSM, we extended the application programming interface with support for point-to-point synchronization primitives `signal` and `wait`. We also extended the barrier mechanism to carry data on the barrier messages. This has many uses, but in our case the motivation is the efficient implementation of *reductions*. To eliminate round-trip communication for each pairwise-shared page following synchronization operations, we augmented our SDSM implementation of point-to-point synchronization to support *selective eager update*, where a compiler-specified set of pages are sent to the synchronization partner along with consistency metadata in anticipation of the waiter's expected future requests. The compiler is conservative, so that no data is pushed but not accessed after the synchronization. The SDSM runtime decides how many data pages can be pushed without overwhelming the communication subsystem.

Opts	Meaning
P	point-to-point synchronization
E	eager update
R	restricted consistency
B	compiler-managed shared buffer

Table 5: Key for row heading abbreviations in performance tables.

Opts	Time (s)	Msg (K)	Comm (MB)
P	918	6,615	17,063
PE	599	617	17,629
PR	918	6,491	16,932
PRE	591	474	17,687

Table 6: NAS-BT class A 16 columnwise computation partitioning

3.10.1 Effectiveness of Integrated Compiler and Run-time Optimizations

To evaluate the effectiveness of our integrated compiler and runtime techniques and understand the interactions among different optimization techniques, we studied the performance of a set of HPF benchmark codes compiled with our SDSM-version of the dHPF compiler and executed on the enhanced TreadMarks SDSM system. Our experiments were performed on an IBM SP2. The SP2 is a distributed memory message-passing machine. Our experimental platform was populated with "high" processor nodes, each consisting of 4 PowerPC 604e processors with 1GB of main memory. Nodes are connected by a multi-layer scalable switching fabric. On each multiprocessor high node, we ran only one process to ensure that all messages were transported across the switch and that there was no contention for the network interface. Unless otherwise stated, all experiments were performed on 16 high nodes with one process active on each.

We only report results for BT, an application benchmark from the NAS 2.0 Benchmark suite. We compare the performance of multiple configurations of compiler and runtime optimizations to ascertain the impact of these optimizations both collectively and individually.

NAS-BT is a large, "whole application" benchmark. Our parallel version was constructed by starting with the sequential version of the BT benchmark from the NAS suite [BHS+95], adding HPF directives, and interchanging a few loops to adjust the pipeline granularity exploited by dHPF. (These changes are described elsewhere [AJMCY98].) For those phases in which computation sweeps along a partitioned dimension the compiler can generate pipelined parallelism. By using point-to-point synchronization, this pipelining becomes a wavefront.

We ran the NAS-BT experiments on the class A problem size using (*, BLOCK) and (BLOCK, BLOCK) data distributions and on 16 processors. The original sequential version runs in 3948 seconds[14].

Table 6 summarizes 16-node runs column-wise 1D partitioning. There is not much false sharing, so the optimizations addressing false sharing give only modest improvements. The message aggregation and latency avoidance of the eager update mechanism does provide significant improvement. The 1D partitioning leads to longer pipelines (which increase serialization) which degrades

[14]Because of hardware differences, the results presented here are not directly comparable with the results presented in [AJMCY98].

Opts	Time (s)	Msg (K)	Comm (MB)
P	926	6,320	14,904
PE	737	3,231	15,052
PR	916	4,674	11,143
PRE	595	754	11,334
PB	1066	6,425	10,689
PEB	852	3,421	10,795
PRB	585	3,252	5,075
PREB	405	247	5,151

Table 7: NAS-BT class A 4x4 computation partitioning. Sequential time is 3948s.

efficiency as more processors are added.

Table 7 summarizes results for (BLOCK,BLOCK) 2D partitioning on 16 processors. False sharing and fragmentation are extensive. Compiler restricted-consistency by itself (PR) hides some of the false sharing, reducing the communication volume by a third, but it does not address communication latency or fragmentation.

The combination of compiler-managed buffer and compiler restricted-consistency (PRB) works as intended to convert highly fragmented pages into falsely shared ones and to then hide the pages from the consistency mechanism. This eliminates two thirds of communication and reduces execution time by over 56%.

Used with any combination of the other mechanisms, eager update decreases communication cost through message aggregation and latency elimination. The best speedup, 10 out of 16, is achieved by applying all of the optimizations (PREB). Once the false sharing and fragmentation problems have been dealt with, the increased parallelism and smaller communication/computation ratio of the 2D partitioning contribute to better overall performance than is achieved using 1D partitioning as the results in Table 6 show.

Our experiments demonstrate that our integrated compiler and runtime support for SDSM can effectively improve the performance of regular applications, including those with loop carried data dependences that require tight-coupling and pipelined codes to parallelize effectively. The combination of compiler-managed restricted consistency in conjunction with compiler-managed communication buffers is very effective at reducing the amount of false sharing and fragmentation for the applications we examined. Our optimizations make scalable multi-dimensional computation partitionings feasible with SDSM.

3.11 Integrated Compiler Support for Tools

For high-level programming languages such as HPF to be useful, tools need to bridge the semantic gap between the high-level source and the generated code. Earlier DARPA-supported research at Rice supported development of the D System programming tools which were integrated with a compiler to support data-parallel programming. In this project we extended that work to support more complex program transformations and provide more detailed user feedback.

The tools work in the dHPF project focused on support for the D Editor, an interactive tool that helps users understand the data-parallel compilation process and the relationship between the code generated and the original source. To support this tool, the dHPF compiler constructs an extensive database during compilation that includes detailed transformation maps to correlate source with output code. These maps enable the user interface to support bi-directional mapping between any fragment in the the source code and the generated code as well as to relate performance to both the generated code and the original source. The source-level program interface also provides information about the the parallelism exploited and communication introduced by the compiler.

Tools support in the dHPF compiler has focused on three major types of support for source-level tools: a transformation hierarchy for tracking program transformations, collecting static analysis results to explain compiler parallelization of the program [ACG+94], and performance-oriented information to map measured performance back to source code [AWMC+95]. During compilation, this information is collected into a program database. The program database serves as a primary information source for several tools with functions including interactive browsing of program analysis and transformation results, instrumenting generated code to collect runtime information, and interpreting runtime information.

The focus of our efforts in this area are summarized briefly below.

- We developed new and more precise techniques to map between the compiler-generated parallel code (the result of a series of transformations) and the source code. In particular, we developed an extended "transformation-tracking mechanism" that computes maps relating the code before and after each transformation at the granularity of individual statements and references. Making this practical in a compiler like dHPF, which makes a very large number of transformations, is a challenging task. We solved this problem as follows.

  First, we developed the concept of a transformation hierarchy, which is a tree-based representation of the transformations performed by the compiler. An internal node in the tree represents a logical transformation performed on the code, while its immediate children represent a sequence of smaller transformations that implement that logical transformation. The root of the transformation hierarchy represents the entire compilation. The leaves of the hierarchy represent the actual low-level manipulations of the program internal representation (e.g., inserting a statement or replacing a reference with another one). To support this tracking mechanism, we modified the program internal representation (an Abstract Syntax Tree) so that multiple versions of program fragments can unambiguously coexist.

  Second, we developed mapping algorithms that traverse the transformation hierarchy in either direction, providing precise information about the set of output AST nodes created from a given source AST node, and vice versa. This enables tools using the program transformation information in the database to accurately map the generated code to the corresponding source code. The performance tool, for instance, uses this mechanism to correctly attribute summaries of the measured dynamic performance to fragments in the original source program as well as to attribute more detailed performance information to fragments in the generated code.

- We redesigned the loop categorization scheme used by the D Editor to provide programmer feedback about compiler-based parallelization of code. Our new strategy supports the dHPF compiler's computation-partitioning model, which is now vastly more general than the one considered previously for loop characterization.

- We developed a tool that uses information in the program database as the basis for instrumenting dHPF-generated programs. The instrumented code uses the AIMS instrumentation library [YSM95] developed by NASA Ames to collect execution traces. Our program instrumenter is capable of instrumenting any message-passing code generated by our compiler, including support for point-to-point message passing, reductions, and broadcasts. An execution of a program instrumented using this tool will collect communication traces that we are able to view graphically using the AIMS XV tool [YSM95]. Our instrumentation tool uses the mapping information in the program database to relate communication in the generated program back to the original HPF program source.

- We added execution analysis support for computing derived statistics from a program's execution trace. These statistics include measures such as the number of invocations for loops and procedures and the time spent in communications other than sending or receiving messages.

- We updated the D Editor user interface to use the dHPF compiler's new program analysis results and compilation information from the program database, and extended the browser was extended to annotate the program with run-time performance about communication and computation using information distilled from AIMS trace information produced by our instrumented programs. Third, display of program run-time performance information has been extended to not only display performance information in terms of the original source, but also for the generated code.

3.12 Other Accomplishments

HPF Benchmark Suite We cooperated with Dr. Hu and Dr. Johnsson and built an HPF benchmark suite (HPFBench) for evaluating the HPF language and compilers on scalable architectures based on their early work on Connection Machine Fortran in Thinking Machine Corp.

The functionality of the HPFBench benchmarks covers linear algebra library functions and application kernels. The motivation for including linear algebra library functions is for measuring the capability of compilers in compiling the frequently time-dominating computations in applications. One motivation for building libraries, in particular in the early years of new architectures, is that they may offer significantly higher performance by being implemented, at least in part, in lower level languages to avoid deficiencies in compiler technology, or in the implementation of compilers and run–time systems. However, though the functionality of libraries is limited compared to that of applications being run on most computers, implementing libraries in low level languages tend to be very costly, and often means that high or even good performance may not be available until late in the hardware product cycle. This in turn implies that following the rapid advances in hardware technology is very difficult, since the older generation hardware often competes successfully with the new generation because of the difference in the quality of software. Thus, it is important to minimize the amount of low level code also in software libraries, and shift the responsibility of achieving high efficiency to the compiler.

In addition to some of the most common linear algebra functions that are frequently occurring in many science and engineering applications, the HPFBench benchmark suite also contains a set of small application codes containing typical "inner loop" constructs that are critical for performance, but that are typically not found in libraries. An example is stencil evaluations in explicit finite difference codes. The benchmarks were chosen to complement each other, such that a good coverage would be obtained of language constructs and idioms frequently used in scientific applications, and for which high performance is critical for good performance of the entire application. Much of the resources at supercomputer centers are consumed by codes used in fluid dynamic simulations, in fundamental physics and in molecular studies in chemistry or biology. The selection of application codes in the HPFBench benchmark suite reflects this fact.

The HPFBench benchmark codes are written entirely in High Performance Fortran 1.0 standard. Tested features cover array operations from Fortran 90, data mapping directives, parallel constructs, and library procedures including scatter operations, parallel prefix operations, and sort opertion from HPF. Details of this work are described in a publication [HJJ$^+$00].

Techniques for Improving the performance of Irregular Applications. As part of this project's research, we worked with irregular applications including molecular dynamics and particle hydrodynamics. Modern computer systems use multi-level memory hierarchies to bridge the ever-widening gap between CPU speed and memory speed. However, multi-level memory hierarchies are typically underutilized by irregular applications because their patterns of indirect accesses have poor locality. We investigated strategies for data and computation reordering to improve utilization of multi-level memory hierarchy for irregular applications. We described and evaluated strategies for data and computation reordering using space-filling curves and introduced multi-level blocking as a new computation reordering strategy for irregular applications. Experiments that applied specific combinations of data and computation reorderings to two irregular programs, overall execution time dropped by a factor of two for one program and a factor of four for the second. One of the codes we used in our experimentation was MAGI, an Air Force particle hydrodynamics code. Our reordering improved end-to-end performance of this code by a factor of 2 on the test case we studied. Details of this work are described in a publication [MCWK99]. Section 6 describes interactions with DoD researchers regarding this technology. Under separate funding from the DOE, we are exploring use of this technology in ASCI applications.

MHSIM: a Multi-level Memory Hierarchy Simulator. For an application code to achieve high performance, it must exploit caches effectively. Most scientific codes in production use were developed for vector proces sors that had no caches. When porting such applications to machines with multiple layers of caches, it is difficult to understand the reasons for poor memory hie rarchy utilization. To address this problem we have developed MHSIM, an integrated instrumentation tool and simulator. MHSIM is designed to

- identify program references causing poor cache utilization,
- quantify cache conflicts, temporal reuse and spatial reuse, and
- correlate simulation results to references and loops in an application program.

The MHSIM simulator and associated instrumentation tool orchestrate a detailed simulation of multi-level memory hierarchies that can help identify the causes res ponsible for poor memory hierarchy utilization and help users achieve a higher fraction of peak performance. The impact and DoD Relevance of this tool is to help DoD users understand how to tune memory hierarchy performance of complex Fortran 77 and Fortran 90 application codes. The MHSIM simulator and instrumentation tool for Fortran programs is available from http://www.cs.rice.edu/ dsystem/mhsim. Supplemental funding for this work was received through the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation. We delivered this tool to Dr. Clay Breshears and Dr. Henry Gabb at **DoD EHRC MSRC** for use by DoD scientists.

3.13 Evaluation of Accomplishments

This project's main objective was to develop compiler analysis and code generation technology that enables machine-independent, data-parallel programs to achieve high performance on a range of scalable parallel architectures for a broad spectrum of scientific applications.

As design and implementation of dHPF began, it was believed that broadening the scope of HPF compilation techniques to handle irregular problems was the most important goal. It soon became clear, though that the principal impediment to commercial success of HPF was low performance, even for the dense matrix computations for which it was designed. While HPF compilers could deliver performance similar to hand-coded applications for a small class of codes that had embarassingly parallel or loosely synchronous computation, more tightly-coupled computations, such as those using ADI integration, were a considerable stumbling block. Even with wholesale rewriting of such codes, the end result was lower performance and scalability than the hand-coded parallel versions.

To address performance concerns across the spectrum of applications, analysis and communication placement techniques implemented in dHPF were designed to support both regular and irregular applications. As we began to work with representative scientific applications such as the NAS parallel benchmarks, it became clear that there were significant shortcomings in compiler technology for regular applications as well. In response, the project began to focus more on trying to make computations for which HPF was designed faster, rather than on broadening the focus of HPF. Without high-quality parallelizations of regular applications, high-level data-parallel programming models would fall by the wayside. Funding cuts of roughly 20% limited our ability to provide complete code-generation support for irregular applications, except in our code generator for shared-memory systems.

As the project funding ended, our goal of transferring compiler technology to deliver performance competitive with hand-coded parallel programs through compiler optimization of data-parallel languages has not been met. While the technology we have developed shows great promise (for example, it can automatically generate extremely sophisticated parallelizations of line-sweep computations such as those in the NAS computational fluid dynamics benchmarks), we have found that having 95% of the technology in place is insufficient to deliver hand-coded performance. Without the last 5% of the optimizations in place, performance and scalability is unacceptable. The lesson for compiler-based parallelizations is that unless everything is perfect, the results are unsatisfactory. We are in the process of completing an overhaul of our communication generation framework to enable sophisticated message aggregation that will improve the efficiency of multi-partitioning. Once this work is integrated back into our main line software, we will distribute the system on the WWW so that others can benefit from the technology we have developed.

4 Project Publications

[1] Ajay Sethi and Ken Kennedy. Resource-Based Communication Placement Analysis In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Compilers for Parallel Computing (LCPC96)*, Lecture Notes in Computer Science 1239, San Jose, California, (August 1996), pages 369-388.

[2] Ajay Sethi and Ken Kennedy. A Communication Placement Framework with United Dependence and Data-Flow Analysis. In *Proceedings of the Third International Conference on High Performance Computing*, India, December 1996. (Also available 1996 International Conference of High Performance Computing) Received best systems paper award, Digital Equipment (India), 1996.

[3] Vikram Adve and John Mellor-Crummey. Advanced code generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, Lecture Notes in Computer Science 1511, Springer-Verlag, 1998.

[4] Bo Lu. Compiling reductions in data parallel programs for distributed memory multiprocessors. Master's thesis, Dept. of Computer Science, Rice University, July 1997.

[5] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code (extended abstract). In *Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 1366, Minneapolis, MN, Springer-Verlag. August 1997.

[6] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in High Performance Fortran. In *Proceedings of SC'97: High Performance Networking and Computing*, San Jose, CA, November 1997.

[7] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[8] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 1998.

[9] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

[10] Vikram Adve, Robert Fowler, Guohua Jin, Ken Kennedy, and John Mellor–Crummey. Advanced Optimization Techniques for HPF. In *Proceedings of the 2nd Annual HPF Users' Group Meeting*, Porto, Portugal, (June 1998).

[11] Gerald Roth and Ken Kennedy. Loop Fusion in High Performance Fortran. In *Proceedings of the 12th ACM International Conference on Supercomputing*, Melbourne, Australia, July, 1998), pages 125-132.

[12] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran compilation techniques for parallelizing scientific codes. In *Proceedings of SC98: High Performance Computing and Networking*, Nov 1998.

[13] Ken Kennedy. Compilers, Libraries, Languages. In *Computational Grids: The Future of High–Performance Distributed Computing, (I. Foster and C. Kesselman, editors)*, Morgan Kaufmann Publishers, Inc., August, 1998.

[14] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[15] Collin McCurdy and John Mellor-Crummey. An Evaluation of Computing Paradigms for N-body Simulations on Distributed Memory Architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[16] Kai Zhang. Compilation and Runtime Optimizations for Software Distributed Shared Memory. Master's thesis, Dept. of Computer Science, Rice University, October, 1999.

[17] Collin McCurdy. An Evaluation of Computing Paradigms for N-body Simulations on Distributed Memory Architectures. Master's thesis, Dept. of Computer Science, Rice University, May, 1999.

[18] Y. C. Hu, G. Jin, S. L. Johnsson, D. Kehagias and N. Shalaby HPFBench: A High Performance Fortran Benchmark Suite *ACM Transactions on Mathematical Software* 26(1), 99–149, March 2000.

[19] Daniel Chavarria-Miranda and John Mellor-Crummey. Toward Compiler Support for Scalable Parallelism using Multipartitioning. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 25-26, 2000

[20] Kai Zhang, John Mellor-Crummey, and Robert Fowler. Compilation and Runtime Optimizations for Software Distributed Shared Memory. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 25-26, 2000

[21] Vikram Adve and Rizos Sakellariou. Compiler synthesis of Task Graphs for Parallel Programs. In *Proceedings of 13th Annual Workshop on Languages and Compilers for Parallel Computing* (LCPC'00), Yorktown Heights, NY, August 2000.

5 Personnel

5.1 Faculty

- Ken Kennedy, Ann and John Doerr Professor of Computer Science, Department of Computer Science, Rice University.
- John Mellor-Crummey, Senior Faculty Fellow, Department of Computer Science, Rice University.

5.2 Research Staff

- Vikram Adve, Research Scientist, Department of Computer Science, Rice University (now Assistant Professor of Computer Science, University of Illinois Champaign–Urbana).
- Robert Fowler, Research Scientist, Department of Computer Science, Rice University.
- Guohua Jin, Research Scientist, Department of Computer Science, Rice University.
- Paul Havlak, Post Doctoral Researcher, Department of Computer Science, Rice University.
- Rizos Sakellariou, Post Doctoral Researcher, Department of Computer Science, Rice University.
- Kathi Fletcher, Research Programmer, Center for Research on Parallel Computation, Rice University.
- Monika Mevenkamp, Research Programmer, Center for Research on Parallel Computation, Rice University.
- Nathan Tallent, Research Programmer, Center for Research on Parallel Computation, Rice University.
- Donald Baker, Research Programmer, Center for Research on Parallel Computation, Rice University.

- Mark Mazina, Research Programmer, Center for Research on Parallel Computation, Rice University.
- Lei Zhao, Research Programmer, Center for Research on Parallel Computation, Rice University.

5.3 Graduate Students

- Dejan Mircevski, Master of Science, Department of Computer Science, Rice University, 1997.
- Bo Lu, "Compiling Reductions in Data-Parallel Programs for Distributed-Memory Multiprocessors," Master's Thesis, Department of Computer Science, Rice University, 1997.
- Mark Anderson, Master of Science, Department of Computer Science, Rice University, 1997.
- Ajay Sethi, "Communication Generation for Data-Parallel Languages," Ph.D. Thesis, Department of Computer Science, Rice University, 1997.
- Qun Wang, Master of Computer Science, Department of Computer Science, Rice University, 1998.
- Nenad Nedeljkovic, Master of Computer Science, Department of Computer Science, Rice University, 1996.
- Kai Zhang, "Compiling for Software Distributed-Shared Memory Systems," Master's Thesis, Department of Computer Science, Rice University, 1999. Currently a Ph.D. Candidate.
- Collin McCurdy, "Efficient Techniques for N-body Simulation on Distributed Memory Architectures," Master's Thesis, Department of Computer Science, Rice University, 1999.
- Chen Ding, Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse", Ph.D. Thesis, Rice University, 2000.
- Li Xu, Ph.D. Candidate, Department of Computer Science, Rice University.
- Qing Yi, Ph.D. Candidate, Department of Computer Science, Rice University.
- Daniel Chavarria-Miranda, Ph.D. Candidate, Department of Computer Science, Rice University.

5.4 Undergraduate Students

- Nathan Tallent, Bachelor of Arts, Rice University, 1998.
- John Campbell, Bachelor of Arts, Rice University, 1998.
- Adam Hunter, Bachelor of Arts, Rice University, 1998
- Trushar Sarang, Bachelor's Candidate, Department of Computer Science, Rice University.

6 Interactions

As part of this project's research, we worked with irregular applications including molecular dynamics and particle hydrodynamics. Dr. M. Ehtesham Hayder, a Rice researcher involved in the DoD High Performance Computing Modernization program, coordinated technology transfer from this project to several DoD sites.

Two topics of special interest to DoD researchers were our work on efficient, parallel, hierarchical methods for computing n-body interaction problems and our work on using data and computation reordering for improving memory hierarchy utilization (and thus overall performance) of irregular applications.

Details of our modifications to Hu and Johnsson's HPF code for simulating n-body interactions using Anderson's hierarchical method (related to the fast multipole method) were provided to DoD labss by Dr. Hayder. A copy of a paper "An evaluation of computing paradigms for n-body simulations on distributed memory architectures" by Collin McCurdy and John Mellor-Crummey (*Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999) describing our work was circulated to Dr. Adam Hughes and Dr. Ruth Pachter (CCM-4, Air Force Research Laboratory, Wright-Patterson Air Force Base) and Dr. Richard Luczak (DoD ASC MSRC), and Dr. Andrew Mark (Army Research Laboratory, Aberdeen Proving Ground). Dr. Hayder also transferred th source code developed in this effort.

At the suggestion of Dr. Hayder, we used MAGI, an Air Force particle hydrodynamics code obtained from David Medina of the US Air Force Research Laboratory as one of the driving applications for our work on irregular applications. The work with Magi addressed issues of program and data structure, techniques for identifying the cause of performance problems, and on solutions to these problems using compiler and run-time technology. Our work with MAGI led to two spinoff technologies.

First, the need to understand MAGI's memory referencing behavior and and its impact on performance led us to develop a simulator for multi-level memory hierarchies. Under supplemental funding (DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation), we delivered this tool, known as MHSIM (http://www.cs.rice.edu/ dsystem/mhsim), to Dr. Clay Breshears and Dr. Henry Gabb at DoD EHRC MSRC for use by DoD scientists.

Second, our experimentation with MAGI led to our development of a new strategy of using space-filling curves as the basis for dynamically reordering data in irregular computations to improve spatial and temporal locality. Our work with MAGI is described in the paper "Improving Memory Hierarchy Performance for Irregular Applications" by John Mellor-Crummey, David Whalley and Ken Kennedy (*Proceedings of the ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999). Dr. Hayder provided copies of this paper to Dr. David Medina, Dr. Adam Hughes, and Dr. Ruth Pachter, all of the Air Force Research Laboratory, Wright-Patterson Air Force Base. Dr. Pachter was also interested in this work for its potential for improving efficiency of molecular dynamics computations.

6.1 Conference Presentations

- Ajay Sethi and Ken Kennedy. "Resource-Based Communication Placement Analysis." Presented at the *Ninth Workshop on Languages and Compilers for Parallel Compilers for Parallel Computing (LCPC96)*, San Jose, California, August 1996.

- Ajay Sethi and Ken Kennedy. "A Communication Placement Framework with United Dependence and Data-Flow Analysis." Presented at the *Third International Conference on High Performance Computing*, India, December 1996. Recieived best systems paper award, Digital Equipment (India), 1996.
- John Mellor-Crummey and Vikram Adve. "Simplifying control flow in compiler-generated parallel code." Presented at the *Tenth International Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, August 1997.
- Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. "Compiling stencils in High Performance Fortran." Presented at *Supercomputing 1997: High Performance Networking and Computing*, San Jose, CA, November 1997.
- Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. Presented at the *Twelth International Parallel Processing Symposium*, Orlando, FL, March 1998.
- Vikram Adve, Robert Fowler, Guohua Jin, Ken Kennedy, and John Mellor–Crummey. "Advanced Optimization Techniques for HPF" Presented at the *Second Annual HPF Users' Group Meeting*, Porto, Portugal, June 1998.
- Vikram Adve and John Mellor-Crummey. "Using integer sets for data-parallel program analysis and optimization." Presented at *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
- Gerald Roth and Ken Kennedy. "Loop Fusion in High Performance Fortran." Presented at the *Twelth ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.
- V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. "High Performance Fortran compilation techniques for parallelizing scientific codes." Presented at *Supercomputing 1998: High Performance Computing and Networking*, Orlando, FL, Nov 1998.
- Collin McCurdy and John Mellor-Crummey. "An evaluation of computing paradigms for n-body simulations on distributed memory architectures." Presented at *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
- John Mellor-Crummey, David Whalley, and Ken Kennedy. "Improving Memory Hierarchy Performance for Irregular Applications." Presented at the *ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.
- Daniel Chavarria-Miranda and John Mellor-Crummey. "Toward Compiler Support for Scalable Parallelism using Multipartitioning." Presented at the *Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 25-26, 2000
- Kai Zhang, John Mellor-Crummey, and Robert Fowler. "Compilation and Runtime Optimizations for Software Distributed Shared Memory." Presented at the *Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 25-26, 2000
- Vikram Adve and Rizos Sakellariou. "Compiler synthesis of Task Graphs for Parallel Programs", Presented at *13th Annual Workshop on Languages and Compilers for Parallel Computing* (LCPC'00), Yorktown Heights, NY, August 2000.

References

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACG+94] V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren, and C. Tseng. Requirements for Data-Parallel Programming Environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, Fall 1994.

[ACIK93] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[AHW94] S.J. Aarseth, M. Henon, and R. Wielen. A comparison of numerical methods for star cluster dynamics. *Astronomy and Astrophysics*, 37:183–187, 1994.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[AJMCY98] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[AL93] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[AMC98a] Vikram Adve and John Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, Lecture Notes in Computer Science 1511. Springer-Verlag, 1998.

[AMC98b] Vikram Adve and John Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[And92] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput*, 13(4):923–947, July 1992.

[AWMC+95] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[Bal90] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[BCG+95] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[BCK+95] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, Santa Barbara, CA, July 1995.

[BCZ92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[BE95] W. Blume and R. Eigenmann. Demand-driven symbolic range propagation. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, OH, August 1995.

[BH86] J. Barnes and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[BHS+95] D. Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[BMN+95] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 704–709, San Francisco, CA, February 1995.

[Bou93] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, June 1993.

[BTC95] Preston Briggs, Linda Torczon, and Keith D. Cooper. Using conditional branches to improve constant propagation. Technical Report CRPC-TR95533, Center for Research on Parallel Computation, Rice University, April 1995.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CL97] S. Chandra and J. Larus. Optimizing communication in HPF programs on fine-grain distributed shared memory. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 100–111, June 1997.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[Dar86] E. Darnell. Special reductions in PFC. Supercomputer Software Newsletter 13, Rice University, October 1986.

[DCZ96] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, October 1996.

[GB92] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication for multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[Ger90] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[GKHS96] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32(2):155–172, February 1996.

[GMS+95] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Physics*, 73:325–348, 1987.

[Han94] Reinhard v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Dept. of Computer Science, Rice University, December 1994. Available as CRPC-TR94494-S from the Center for Research on Parallel Computation, Rice University.

[Har77] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[Hav94] Paul Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[HBB+95] J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal of Digital Equipment Corp.*, 7(3):5–23, Fall 1995.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[HJ96] Y. Charlie Hu and S. Lennart Johnsson. Implementing $o(n)$ n-body algorithms efficiently in data-parallel languages. *Scientific Programming*, 5(4):337–364, 1996.

[HJJ+00] Y. C. Hu, G. Jin, S. L. Johnsson, D. Kehagias, and N. Shalaby. Hpfbench: A high performance fortran benchmark suite. *ACM Transactions on Mathematical Software*, 26(1):99–149, March 2000.

[HJT97] Y. Charlie Hu, S. Lennart Johnsson, and Shang-Hua Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[HKT92] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[HKT93] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[HTK98] Hwansoo Han, Chau-Wen Tseng, and Pete Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 26(5):591–612, October 1998. Invited paper from LCPC'97.

[Joh86] Harold Johnson. Data flow analysis of 'intractable' imbedded system software. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 109–117, 1986.

[KA97] K. Kennedy and R. Allen. *Advanced Compilation for Vector and Parallel Computers.* Morgan Kaufmann Publishers, San Mateo, CA, 1997.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, April 1996.

[KNS95] K. Kennedy, N. Nedeljković, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, La Jolla, CA, June 1995.

[LC91] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[LCD+96] H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Software distributed shared memory support for irregular applications. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 48–56, June 1996.

[LMC98] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, FL, March 1998.

[MCA97] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code (extended abstract). In *Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 1366, Minneapolis, MN, August 1997. Springer-Verlag. A full version of this paper was selected for publication in a special issue of the *International Journal of Parallel Programming.*

[MCA98] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 26(5), 1998.

[McC99] Collin McCurdy. Efficient techniques for n-body simulation on distributed memory architectures. Master's thesis, Dept. of Computer Science, Rice University, 1999. Forthcoming.

[MCWK99] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 425–433, Rhodes, Greece, 1999.

[MMC99] Collin McCurdy and John Mellor-Crummey. An evaluation of computing paradigms for n-body simulations on distributed memory architectures. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[MV89] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[Nai92] V. Naik. Scalability issues for a class of CFD applications. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[NNN93] N.H. Naik, V. Naik, and M. Nicoules. Parallelization of a class of implicit finite-difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1):1–50, 1993.

[NQ93] N. Nedeljkovic and M. J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice and Experience*, 5(4):257–268, June 1993.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[Sag94] Hans Sagan. *Space-Filling Curves.* Springer-Verlag, New York, NY, 1994.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, Chichester, Great Britain, 1986.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[SKN96] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

[SOG94] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[TP95] Peng Tu and David Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[Tse93] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[Van93] R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Proceedings of Supercomputing 1993*, pages 102–111. IEEE Computer Society Press, 1993.

[vDSP96] Kees van Reeuwijk, Will Denissen, Henk Sips, and Edwin Paalvast. An implementation framework for hpf distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):897–914, September 1996.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[WS93] M. Warren and J. Salmon. A parallel hashed-octtree n-body algorithm. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[YSM95] J. C. Yan, S. R. Sarukkai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the aims toolkit. *Software—Practice and Experience*, 25(4):429–461, April 1995.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[Zha99] Kai Zhang. Compilation and runtime optimizations for software distributed shared memory. Master's thesis, Dept. of Computer Science, Rice University, October 1999.

*U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10059

DISTRIBUTION LIST

addresses	number of copies
JOSEPH CAROZZONI AFRL/IFTB 525 BROOKS RD ROME NY 13441-4505	1
RICE UNIVERSITY POB OX 1892 - MS 16 HOUSTON TX 7751-1892	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL US ARMY SPACE & MISSILE DEF CMD P.O. BOX 1500 HUNTSVILLE AL 35807-3801	1
COMMANDER, CODE 4TL000D TECHNICAL LIBRARY, NAWC-WD 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6100	1
CDR, US ARMY AVIATION & MISSILE CMD REDSTONE SCIENTIFIC INFORMATION CTR ATTN: AMSAM-RD-OB-R, (DOCUMENTS) REDSTONE ARSENAL AL 35898-5000	2
REPORT LIBRARY MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1
ATTN: D'BORAH HART AVIATION BRANCH SVC 122.10 FOB10A, RM 931 800 INDEPENDENCE AVE, SW WASHINGTON DC 20591	1
AFIWC/MSY 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
ATTN: KAROLA M. YOURISON SOFTWARE ENGINEERING INSTITUTE 4500 FIFTH AVENUE PITTSBURGH PA 15213	1
USAF/AIR FORCE RESEARCH LABORATORY AFRL/VSOSA(LIBRARY-BLDG 1103) 5 WRIGHT DRIVE HANSCOM AFB MA 01731-3004	1
ATTN: EILEEN LADUKE/D460 MITRE CORPORATION 202 BURLINGTON RD BEDFORD MA 01730	1

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202